KTH Computer Science
and Communication

Logics for Information Flow Security: From Specification to Verification

MUSARD BALLIU

Doctoral Thesis in Computer Science
Stockholm, Sweden 2014

TRITA-CSC-A-2014:13
ISSN-1653-5723
ISRN KTH/CSC/A–14/13–SE
ISBN 978-91-7595-259-8

KTH CSC TCS
SE-100 44 Stockholm
SWEDEN

Akademisk avhandling som med tillstånd av Kungl Tekniska högskolan framlägges till offentlig granskning för avläggande av teknologie doktorsexamen i datalogi fredagen den 3 oktober 2014 klockan 14.00 i Kollegiesalen, Kungl Tekniska Högskolan, Brinellvägen 8, Stockholm.

Tryck: E-print

Abstract

Software is becoming increasingly ubiquitous and today we find software running everywhere. There is software driving our favorite game application or inside the web portal we use to read the morning news, and when we book a vacation. Being so commonplace, software has become an easy target to compromise maliciously or at best to get it wrong. In fact, recent trends and highly-publicized attacks suggest that vulnerable software is at the root of many security attacks.

Information flow security is the research field that studies methods and techniques to provide strong security guarantees against software security attacks and vulnerabilities. The goal of an information flow analysis is to rigorously check how sensitive information is used by the software application and ensure that this information does not escape the boundaries of the application, unless it is properly granted permission to do so by the security policy at hand. This process can be challenging as it first requires to determine what the applications security policy is and then to provide a mechanism to enforce that policy against the software application. In this thesis we address the problem of (information flow) policy specification and policy enforcement by leveraging formal methods, in particular logics and language-based analysis and verification techniques.

The thesis contributes to the state of the art of information flow security in several directions, both theoretical and practical. On the policy specification side, we provide a framework to reason about information flow security conditions using the notion of knowledge. This is accompanied by logics that can be used to express the security policies precisely in a syntactical manner. Also, we study the interplay between confidentiality and integrity to enforce security in presence of active attacks. On the verification side, we provide several symbolic algorithms to effectively check whether an application adheres to the associated security policy. To achieve this, we propose techniques based on symbolic execution and first-order reasoning (SMT solving) to first extract a model of the target application and then verify it against the policy. On the practical side, we provide tool support by automating our techniques and thereby making it possible to verify programs written in Java or ARM machine code. Besides the expected limitations, our case studies show that the tools can be used to verify the security of several realistic scenarios.

More specifically, the thesis consists of two parts and six chapters. We start with an introduction giving an overview of the research problems and the results of the thesis. Then we move to the specification part which relies on knowledge-based reasoning and epistemic logics to specify state-based and trace-based information flow conditions and on the weakest precondition calculus to certify security in presence of active attacks. The second part of the thesis addresses the problem of verification of the security policies introduced in the first part. We use symbolic execution and SMT solving techniques to enable model checking of the security properties. In particular, we implement a tool that verifies noninterference and declassification policies for Java programs. Finally, we conclude with relational verification of low level code, which is also supported by a tool.

Sammanfattning

Programvara har blivit mer och mer närvarande i samhället, och vi hittar den idag i stort sett överallt. Program driver våra favoritspel och körs av webbportalen där vi läser morgonnyheterna eller bokar vår semester. Den stora spridningen gör att program blir en enkel måltavla för uppsåtligt utnyttjande, eller, i bästa fall, ofta beter sig felaktigt. Trender och omskrivna incidenter pekar på att sårbarheter i program utgör ingången till många attacker på datorsystem.

Informationsflödessäkerhet är ett forskningsfält som studerar metoder och tekniker som ger starka garantier mot förekomsten av attackvägar och sårbarheter. Målet med en informationsflödesanalys är att rigoröst följa hur känslig information används av ett program, och att säkerställa att informationen inte läcker utanför fastställda ramar om så inte har medgetts av en given säkerhetspolicy. Den här processen kan vara utmanande, eftersom den först kräver att ett programs säkerhetspolicy fastställs och sedan att en mekanism tillhandahålls som säkerställer att policyn följs i programmet. I den här avhandlingen addresserar vi problemet att specificera en (informationsflödes) policy och se till att den efterföljs genom att använda formella metoder, speciellt logiker och språkorienterad analys, samt verifikationstekniker.

Avhandlingen bidrar till att föra forskningen inom informationsflödessäkerhet framåt på flera sätt, både teoretiska och praktiska. På policyspecifikationssidan tillhandahåller vi ett ramverk som möjliggör resonemang om informationssäkerhetsvillkor i termer av kunskapsteoretiska begrepp. Ramverket åtföljs av logiker som kan användas för att uttrycka precisa säkerhetspolicys syntaktiskt. Vi undersöker också samspelet mellan konfidentialitet och integritet för att garantera säkerhet när aktiva attacker kan förekomma. På verifikationssidan tillhandahåller vi flera symboliska algoritmer för att effektivt kontrollera huruvida ett programs beteende är inom ramarna för en associerad säkerhetspolicy. Vår ansats är att använda tekniker baserade på symbolisk exekvering och första ordningens slutledning (SMT-lösning) för att först extrahera en modell av målprogrammet och sedan verifiera modellen mot policyn. På den praktiska sidan tillhandahåller vi verktygsstöd genom att automatisera vår ansats och därmed möjliggöra verifikation av program skrivna i Java eller maskinkod för ARM-processorer. Förutom de förväntade begränsningarna, visar våra fallstudier att verktygen kan användas för att verifiera säkerhet i flera realistiska scenarier.

Mer specifikt består avhandlingen av två delar och sex kapitel. Vi inleder med en introduktion som ger en överblick av forskningsproblemen och resultaten i avhandlingen. Vi går sedan vidare till en specifikationsdel, som utgår från kunskapsteoretiska begrepp och epistemiska logiker för att möjliggöra specifikation av tillståndsbaserade och spårbaserade informationsflödesvillkor, och från en kalkyl för svagaste förvillkor för att möjliggöra certifiering av säkerhet när aktiva attacker kan förekomma. Den andra delen av avhandlingen adresserar problemet med verifiering av säkerhetspolicys som introduceras i den första delen. Vi använder symbolisk exekvering och SMT-lösningstekniker för att möjliggöra modellprovning av säkerhetsegenskaper. Specifikt implementerar vi ett verktyg som verifierar störningsfrånvaro och avklassifieringspolicys för Java-program. Vi avslutar med beskriva relationell verifiering av lågnivåkod, som också stöds i ett verktyg.

Acknowledgements

It has been a long journey with many ups and downs, but here I am at the end. I would like to take this opportunity to thank all those people who contributed to the completion of this thesis.

I am truly thankful to my advisor Mads Dam for his excellent guidance, patience and caring. He always allowed me to pursue my research interests, teaching me new things and providing a stimulating environment for doing research. I am grateful to Mads also for many personal advices when I moved to Sweden and for the wonderful sailing trips in the archipelago. Mads, you have been, and will continue to be, a role model for me.

I consider myself lucky to have collaborated and coauthored papers with bright researchers such as Mads Dam, Gurvan Le Guernic, Roberto Guanciale and Isabella Mastroeni. They have all been a great source of inspiration for my research.

I would like to thank everybody, present and past members, of the TCS department at KTH. It has been a great pleasure to share a dynamic working environment with you. Special thanks are due to Andreas, Björn, Cenny, Dilian, Douglas, Emma, Gurvan, Hamed, Karl, Lukáš, Ola, Oliver, Pedro, Roberto, Sangxia, Shahram, Siavash, Stefan and Torbjörn many discussions, activities and beers. Earlier drafts of the thesis have benefited useful feedback from Mads Dam, Dilian Gurov, Roberto Guanciale, Elton Kasmi, Michael Minock and Karl Palmskog. Thanks for your help.

I am grateful to Andrei Sabelfeld and to the Language-based Security group at Chalmers for considering me, as Andrei put it, a brother PhD student from the sister university of KTH. Special thanks go to Andrei for the many activities we have enjoyed together.

My PhD work has benefited a lot from discussions with Steve Chong, Roberto Giacobazzi, Gurvan Le Guernic, Roberto Guanciale, Dilian Gurov, Johan Håstad, Isabella Mastroeni, Alejandro Russo, Andrei Sabelfeld, Dave Sands, Fausto Spoto, Ola Svensson and Luca Viganò. Further thanks go to my grading committee members David Naumann, Dave Clarke, Bernd Finkbeiner and Marieke Huisman for accepting to evaluate my work and providing useful feedback.

Life in Stockholm has been rewarding, lots of fun and many friendships. I will never forget the nights out with Kristaps Dzonsons, Pedro Gomes and Dilian Gurov. Dilian, thanks for being a good friend and for always believing in me as a researcher. Kristaps, Pedro, I know I can always count on you.

Special thanks go to Ledio Koshi, Ferdinand Laci and their families for being a constant support and always making me feel home. This experience wouldn't have been the same without my Albanian friends, Alban and Dorian. Thank you guys for the wonderful time, the dream goes on. Alban, you are the best friend I could wish for, I learned from you more than you can imagine.

My greatest gratitude goes for my Mom and Dad for their constant encouragement, support and unconditional love. Words are not enough to thank Besa for what she has gone through just to be with me. Thank you for loving me and being so close, despite the distance. Luv u!

Contents

Chapter 1

Introduction

For better or worse, the advent of the Information Age has certainly marked a new revolutionary era of humankind. This is best reflected in the way our everyday life strongly depends on information and communication technologies (ICT). Modern buzzwords such as e-government, e-business or e-health have made their way into our common language to refer to any government, business or healthcare process that is conducted in a digital form via the Internet. In many directions, this increasing connectivity using computers and networks has provided more goods and improved people's lives. But as usual, there is no free lunch, it all comes at a price [22, 170, 140, 109].

In April 2014 a security firm called Codenomicon and a Google researcher independently discovered a security flaw, dubbed Heartbleed, in an open-source cryptographic software library (OpenSSL) that is used by an estimated two-thirds of web servers [4, 5]. OpenSSL is behind many secure communication routines over the Internet and Heartbleed can be exploited easily to leak encryption keys, passwords, email and financial data, and seriously compromise the security of everyone who has access to a network. Heartbleed results from improper input validation, known as buffer over-read, an attack where the software reads more data than it should be allowed to. Figure 1.1 depicts a normal scenario and an attack scenario to exploit the bug [13]. A normal "Heartbeat" request would require a client to send a message, consisting of a payload, typically a text string (e.g. *blah*), along with the payload's length (e.g. 4). The server then must send the exact same payload back to the client. However, the client, either accidentally or maliciously, can make a request consisting of the same payload as in the normal scenario (e.g. *blah*), but with a bigger payload's length (e.g. 40004). As a result, the message returned consists of the payload, followed by whatever else happened to be in the allocated memory, potentially sensitive information.

The estimated cost of Heartbleed is 500 million dollars as a starting point [3]. Even worse, after decades of research and experience in computer security, many experts argue that most of the existing tools would have failed to discover the bug

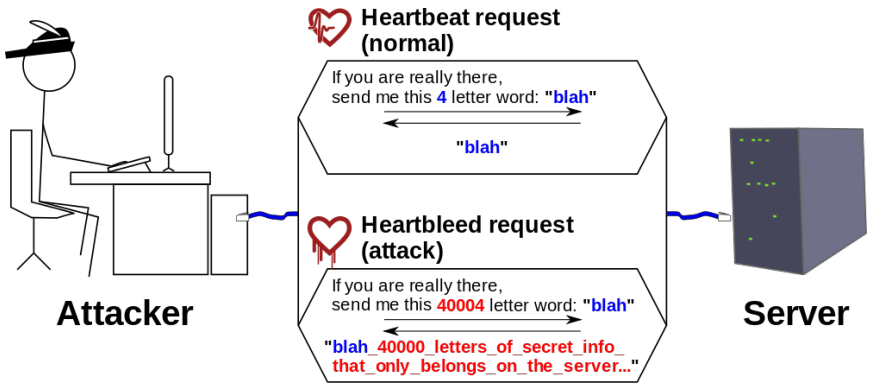


Figure 1.1: Heartbleed bug explanation

[221]. Unbelievable! Heartbleed, one of the most dangerous security bugs ever, calls for serious reflection by everyone, in research and industry.

The increase of the number of software security threats over the years might seem surprising at first. Nevertheless, there are good reasons to believe that, unless the approach to security becomes more formal and systematic, this trend will continue. First and foremost, the technological shift from the old mainframe, where many people shared one computer, to the personal computer where everyone has his own computer, is now transitioning, through distributed computing, towards the ubiquitous computing model where lots of computers will share each of us. This increasing dependence on the Internet, which originally was not designed with security in mind, and the unavoidable need to exchange and share information make it easy to distribute malicious code and give rise to new attack vectors. Moreover, modern information systems are tremendously complex and heterogeneous, and, as US President's IT Advisory Committee put it [143], we simply do not know how to design and test software systems with millions of lines of code in the same way that we can verify whether a bridge or an airplane is safe [150]. For instance, the Android operating system consists of 12 million lines of code including 3 million lines of XML, 2.8 million lines of C, 2.1 million lines of Java, and 1.75 million lines of C++ [12]. Not to mention that today software is extensible and evolves frequently. Many desirable features require embedding code from potentially untrusted parties and allowing dynamic software updates across different execution platforms. It is clear that we have to live with these trends and devise new methods and techniques that allow both trusted and untrusted software to share the same space, without compromising security. To achieve this goal, we need to define precisely what *security policies* to enforce in the system and what *mechanisms* to use for enforcing these policies. This thesis addresses these issues and provides solutions for both.

The remainder of this chapter is organized as follows. Section 1.1 gives an overview of information security requirements, with emphasis on how these re-

quirements can be approached in a formal manner. It also motivates why the existing solutions are not satisfactory. Section 1.2 introduces information flow security which is the main topic of this thesis. It discusses the general context and various solutions and complications that may arise. Section 1.3 focuses on the state of the art, including some historical background and recent developments. It then gives a quick taste of the research problems and the solutions proposed in the thesis. Section 1.4 starts with a formal background on the overall specification and verification approach, it goes on to describe briefly each of the included papers, and concludes with a statement of the author's contributions and final remarks.

1.1 Information Security: Policy, Mechanism, Adversary

Broadly speaking, information security requirements, or security policies, focus on confidentiality, integrity and availability of information, often referred to as CIA requirements [209]. Confidentiality policies assure that sensitive information is not made available to unauthorized users by restricting who is able to learn the private information. Integrity policies assure that information is not changed by unauthorized users by restricting who is able to create and modify the trusted information. Availability policies assure that systems work promptly and services are not denied to authorized users. In general, security requirements consist of an amalgamation of CIA requirements, as shown in Fig. 1.2. For example, the Heartbleed security bug described in Fig. 1.1 is due to a missing bounds check (integrity violation) which is used to exploit a buffer over-read and, as a consequence, to learn sensitive data (confidentiality violation).

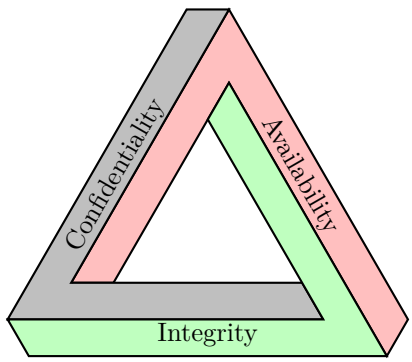


Figure 1.2: CIA Information Security Requirements

A security mechanism consists of a set of methods and techniques which are used to verify and enforce the information security requirements of a system. Traditionally, the mechanisms deployed to secure computer systems comprise various forms of access control and cryptographic techniques. Access control is a way of limiting access to resources or information only to authorized users. For instance,

a user who uploads a picture to a social network may use access control to specify that only his friends are allowed to view the picture. The evolution of access control policies is a good example showing that, as systems become more and more open, the resulting increase of distrust between principals is accompanied by more complex and fine-grained policies. Indeed, in the early days software would run on single-user machines with direct console access free to do anything; later, with the advent of multi-user machines, per-user access control was enforced to accommodate multiple users. Next, privileges were gradually introduced at process level to reflect the fact that not all processes could be trusted equally, even when executing on behalf of the same user. For instance, an Apache Web Server typically starts up the main server process, httpd, as root and then spawns new httpd processes that run as low privilege to handle the Web requests [136]. More recently, richer forms of access control, for instance *sandboxing* [124], were introduced to constrain untrusted parts of the same program, e.g. third-party code, to execute in isolation with restricted access permissions. And this is not the end of the story. What about all those phone apps that ask for network and storage permissions? The security mechanisms and the security policies we study in this thesis provide even stronger security guarantees, which, as we shall see, are needed to properly secure modern applications.

The adequacy of a security mechanism with respect to a security policy is strongly dependent on the adversary model. This model defines who we are protecting against in terms of the capabilities of the adversary. For instance, in network security the Dolev-Yao adversary model considers an active intruder with full network control (i.e., one who intercepts, reads and modifies the network traffic), unable to break cryptography [105]. Real world protocols, however, show that realistic adversaries can compromise session keys or randomness, hence computational models of adversaries that limit the full trust in cryptographic primitives have been considered to cope with these issues. Similarly, access control tacitly assumes that the adversary can not tamper with the enforcement mechanism itself, otherwise the security will be broken. Ideally, we would like to prove our systems secure against the most powerful adversaries, however in many cases this is neither needed nor possible. The real challenge is then to determine the right level of the adversary model, the security policy and the security mechanism that make it "the most difficult" to break security. In general this process may require a more elaborate analysis of both technical and non-technical aspects, for instance risk assessment, usability, social engineering or laws in force. However, as a chain is only as strong as its weakest link, our analysis caters for security at the application level, which is what attackers target the most nowadays. Fig. 1.3 illustrates the basic ingredients needed to define the security analysis requirements. In this thesis we formalize each of these components using rigorous mathematical and logical methods, which allow us to precisely define what security means for a given system.

It is well known that standard security mechanisms such as access control, cryptography or firewalls fall short in preventing modern malware from affecting computing systems [197, 219]. The fundamental reason is that these mechanisms only

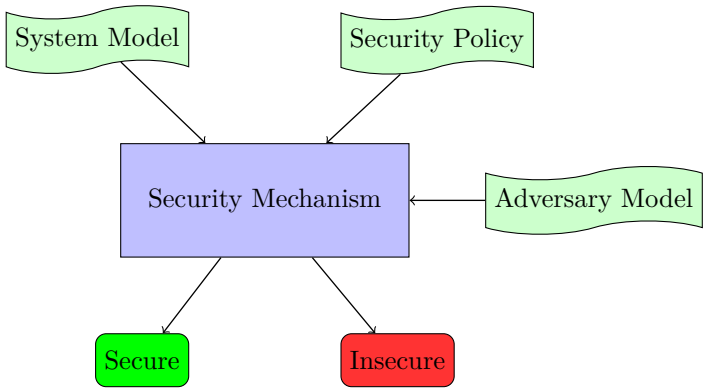


Figure 1.3: Security Analysis Requirements

constrain the access to the information, i.e. what data one can read or write, and do not control how this information is used by the computation, for instance where the information is allowed to flow. Namely, once access to a piece of information is granted, there is nothing preventing it from being propagated through error or malice to an untrusted site. Recent trends show that these kind of scenarios, where trusted and untrusted programs need to share the same execution environment and access sensitive data, arise in many applications. Examples can be found in a browser, where code from different providers needs to be integrated on the same web page, giving rise to a variety of code injection attacks [140]. Or, in a smartphone where apps written by potentially untrusted developers are used by millions of users, giving rise to different security and privacy issues [231, 117, 83]. Or, in the huge codebase of an OS where different types of low level bugs, e.g. buffer overflows, can be exploited to inject viruses, trojans and the like [17, 111].

To better illustrate this point, consider a user, Besa, who travels a lot and very often needs to book a hotel. Besa decides to install an app, BookHotel, which will help her to book a room at the nearest hotel at a reasonable price. Among other permissions, the app requires access to the network (to communicate with the bank), access to the location (to find the nearest hotel) and access to the credit card number (to finalize the booking). To function correctly, the app must have access to all such permissions. However, Besa would like her credit card number to only be sent to the bank and not to the app developers or to Google. Nor does she want her current location to be disclosed to the hotel website. Unfortunately, this type of security policies can not be enforced by access control as they regard the way the information is propagated and used by the app program.

In all these scenarios, the root cause of the security problem is the flow of sensitive/untrusted information to/from unauthorized agents in the system. Studies and statistics show that the majority of security failures are due to security violations at the application level [10, 140]. This calls for new security mechanisms that track

information flow dependencies in the executing program and ensure that they do not violate the security policy. In the security literature, this approach is known as information flow security [122]. Information flow security applies the well-known principle of end-to-end design to certify and build trustworthy systems. In particular, it can provide end-to-end security guarantees by means of a formal analysis or other validation techniques, showing that the system *as a whole* enforces the security requirements of its users. In the example above, an information flow policy would state that the credit card number is only given to the hotel website, while the location information is only used by Google. Information flow control (IFC) constitutes a very promising countermeasure against the proliferation of security attacks that go beyond access control, by ensuring strong and provable security guarantees of the underlying system. However, IFC necessitates an analysis of the target system as a whole, which poses both theoretical and practical challenges [226, 197]. Addressing some of these challenges is the main topic of this thesis.

1.2 Information Flow Control to the Rescue

The ultimate goal of information flow control is to establish confidentiality and integrity properties of code executing on real computers. For confidentiality, sensitive information must be prevented from flowing to public destinations, and dually, for integrity, untrusted information must be prevented from affecting, or flowing to, trusted destinations. Availability is usually ignored by information flow analysis as it is can be studied using other methods.

A rigorous information flow security analysis requires to follow the recipe in Fig. 1.3 and answer the fundamental question:

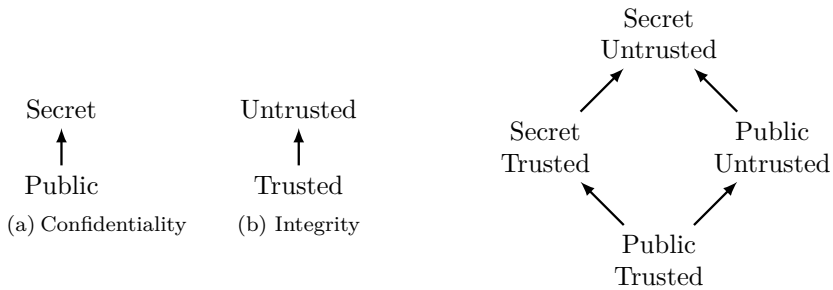
What constitutes a secure system?

A possible answer can be given by leveraging formal methods and using mathematical constructions to state the information flow properties of the system. For instance, the system model can be represented as a state transformer, which produces a set of executions, also called behaviors. The security policy is then defined as a property that needs to be entailed by the system model. The adversary, i.e. the attacker, is normally assumed to be able to partially observe system behaviors, for instance by observing part of the execution state or other system events. In addition, the system model is considered public knowledge.

Broadly, an information flow security policy is defined on a mutli-level security lattice [99], which provides a security classification, or labeling, of the data[1]. Fig. 1.4 illustrates three security lattices for confidentiality, integrity and a combination of the two. The confidentiality lattice in Fig. 1.4a labels the data as *high*, i.e. secret, or *low*, i.e. public, and the attacker can be assumed to observe the data labeled as low. Similarly, the integrity lattice in Fig. 1.4b labels the data as

[1]Depending on the context, the term *data* may refer to users, processes, program states or any other events of interest.

trusted or *untrusted* and the attacker can be assumed to control the data labeled as untrusted. The structure of the security lattice determines the set of allowed and disallowed flows of information. In particular, the direction of the arrows in Fig. 1.4 indicates the allowed flows of information. This relation is later enforced by the security mechanism to ensure that the information contained in the high data is not being leaked through the low data. For integrity, this implies that the information originating from untrusted data does not affect the trusted data. A more complex lattice, shown in Fig. 1.4c, allows to label the data with a confidentiality level and an integrity level.



(a) Confidentiality (b) Integrity (c) Product Lattice

Figure 1.4: Security Lattices

Fig. 1.5 depicts a system model, where the source nodes denote high security data and the sink nodes denote low security data. As the system model is public knowledge, the attacker knows that all four executions are possible and can observe the low security data once the execution has reached a sink node.

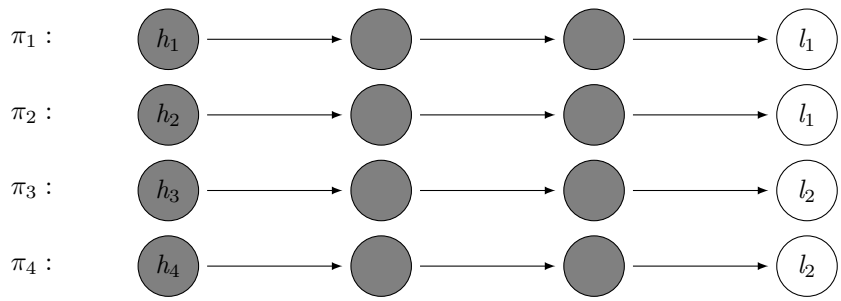


Figure 1.5: A System Model

The semantic (or extensional) security condition is then introduced to determine whether the system model adheres to the security policy. The semantic security condition is important as it defines the baseline against which the correctness of the security mechanism can be validated. In general, the shape of semantic security

condition depends on the power of the attacker, i.e. on the observations she is assumed to be able to make, and on the information that needs to be protected. This gives rise to different types of conditions which target different flavors of security policies and system models. *Noninterference* is probably the most well-known semantic security condition in the information flow literature [122]. Noninterference, as depicted in Fig. 1.6, states that the high/untrusted inputs of the system should not affect the low/trusted outputs of the system. For confidentiality, this means that for any pair of executions, starting from the same low inputs, the resulting final states contain the same low outputs, regardless of the high inputs. As an example, consider the system model in Fig. 1.5, where h_i:s are high inputs and l_i:s are low outputs. The model does not satisfy the noninterference condition. If we consider the pair of executions (π_2, π_3) (they start with the same low inputs, since there are none), the resulting public outputs are different, namely (l_1, l_2). In fact, an attacker who knows the system model and observes the output l_1 (resp. l_2) will be able to learn that the secret input was either h_1 or h_2 (resp. either h_3 or h_4). These kinds of *covert* information flows are ruled out by IFC. Other information flow security conditions, which we discuss later in the thesis, account for more expressive security policies and computational models addressing issues related to compositionality, concurrency or tractability [35].

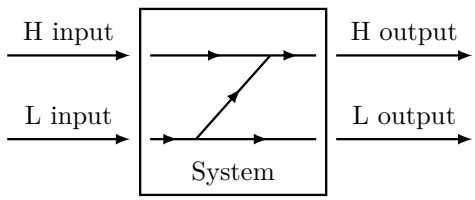


Figure 1.6: Noninterference Condition

The security mechanism embodies methods and techniques for verification and enforcement of the information flow properties, ranging from syntactic to semantic and from static to dynamic approaches. In particular, the semantic security condition is needed to formally prove *soundness* (only secure system models are accepted by the mechanism) and *precision* (what secure systems are incorrectly ruled out due to mechanism incompleteness) of a given security mechanism. This connection is important as it ensures that the mechanism actually guarantees the security policy in the sense of the semantic security condition.

1.2.1 Language-based Information Flow Security

Information flow security, as presented in the previous subsection, is an interesting conceptual model which can be used to reason about expressive security properties of system models. However, systems are more complex than the abstract models considered so far. Typically, they consist of software programs written in a

programming language with a well-defined syntax and a formal semantics, that can be executed on real computers. Language-based (information flow) security is the research area that combines programming languages and computer security techniques to certify information flow properties of software systems [197]. In particular, one can leverage existing program analysis and verification techniques to formally analyze and enforce the security requirements of the program as a whole. This is desirable since it is the program code which is ultimately run on the execution platform, hence it becomes crucial to prove that the security requirements are explicitly supported by the program implementation and that the enforcement mechanism provably certifies this. The Heartbleed bug in Fig. 1.1 is again an example of how an implementation error can seriously compromise security, despite the fact that abstract models, e.g. the design, of the SSL protocol has been extensively verified as flawless.

It is worth pointing out that complete security is an unrealistic and unachievable goal and language-based security alone is insufficient to prevent lower level attacks from breaking into the hardware, measuring power consumption or exploiting other architecture-dependent features such as caches and pipelines. Indeed, a complete security analysis would require pervasive formal verification of both software and hardware, including compilers, linkers, operating systems and other components. This is infeasible in first place due to computability and complexity reasons. Nevertheless, as mentioned earlier, statistics [10] show that the majority of security attacks occur at the software level, hence language-based approaches can significantly contribute to eliminate these attacks and increase our confidence on the software we run on our machines [147, 155].

Schneider et al. [204] argue that language-based security techniques are now needed to implement the classical security principle of *least privilege*. The principle of least privilege states that each agent should be accorded the minimum access necessary to accomplish its task throughout the execution. The shift from coarse-grained per-user access control policies to fine-grained per-application information flow policies requires a departure from traditional OS-like enforcement towards novel approaches that instantiate this principle. In particular, an information flow policy can characterize the secure behaviors of the application, hence define the least privileges needed by that application to function securely. For instance, an information flow policy would constrain the BookHotel app mentioned earlier to only send the credit card number to the hotel website and the location information to Google, and thus define the least privileges of the application.

The baseline for language-based security is the program code, provided in terms of source code, bytecode or even machine code. The attacker is now assumed to know the program code and to observe or modify the runtime behavior through predefined communication primitives. The communication primitives, depending on the program under consideration, can be shared program variables, API methods, channels, CPU registers, memory locations or other. The primitives are associated with security labels and in general involve a security lattice [99], as shown in Fig. 1.4.

The knowledge of the program code, the low and/or untrusted primitives and the execution context may give rise to mechanisms that either maliciously or unintentionally transfer sensitive information to the attacker. These mechanisms are referred to as *information flow channels*. In this thesis we leverage language-based techniques to enforce information flow policies with respect to information flow channels, which we describe below.

1.2.2 Information Flow Channels

Information flow channels may arise for several reasons in many applications. What makes these channels potentially dangerous and their verification challenging is the knowledge of the execution context, which may allow an attacker to combine this knowledge with the public data released by the program and learn sensitive information. As an extreme example, consider a conference management system (CMS) used in the scientific community to review submissions to conferences. Suppose that the CMS sends a notification email to each of the authors when the paper decision has been made. If the paper is accepted, the system sends to the authors another email with additional information. Then, anyone who observes the email traffic can see those emails being sent and learn whether an author got the paper accepted or not[2]. In this subsection, we give an overview of the types of channels that may be exploited by attackers with different capabilities or that may be introduced unintentionally by the programmers.

The easiest way to leak sensitive information is by directly transmitting high data to low data, known as *explicit* flows. For instance, a program can embed the code snippet `send:=pwd`, which directly assigns a high variable `pwd` containing a password to a low variable `send`, which is later used to send information over the network. The knowledge of the program code can be used to reveal sensitive information through the control structure of the program, known as *implicit* flows [100]. The program in Fig. 1.7 contains no explicit flows, however a positive password value is indirectly copied to the public variable `send`. Hence, an attacker who knows the program code and observes the final value of variable `send`, can reveal the entire password. The reader may have noticed that the information leakage is exponential in the size of the secret `pwd` and thus unrealistic for big-size secrets. However, using standard techniques, the implicit flows can be magnified by loops and turn a one-bit leak into an n-bit leak in polynomial time in the size of the secret, cf. [195]. The main goal in these examples is to give a flavor of the different types of channels in a simple manner.

A more powerful attacker, which is able to inject code in a program, can give rise to an *injection* flow. Again, consider the code snippet `tmp:=0;`[•]`;send:=tmp`, which only contains low security variables, and thus it is secure if [•] is replaced with `skip` . However, an attacker able to inject `tmp:=pwd` at [•] can reveal the

[2]This channel was recently experienced by the thesis author, who luckily received two such emails. On a side note, the previous sentence leaks information in the context of this thesis. We challenge the reader to find out what.

```
send:=0;
while (pwd>0) {
     send++; pwd--;
}
```

Figure 1.7: Implicit Flow

password through variable `send`, as it can be observed from the resulting program `tmp:=0;tmp:=pwd;send:=tmp`. These channels may arise in web scenarios where code from different providers may be included in the same web page, for instance using the Javascript language.

More complex information flow channels may arise when high security data affect the timing behavior of the program [16]. A typical example of leakage through a (external) *timing* channel is the modular exponentiation routine used in cryptographic algorithms such as RSA [148]. Consider the program in Fig. 1.8 where all

```
res:=1;
for (i:=0; i<k.length; i++) {
   if (k[i]) {tmp := res*M mod n;}
   else      {tmp := res;}
   res := tmp*tmp;
}
```

Figure 1.8: External Timing in Modular Exponentiation $M^k \ mod \ n$

variables are high. An attacker who is able to measure the running time of this program can still leak the entire secret key `k`, essentially, by exploiting the fact that the instructions in the conditional branch take different amounts of time to execute, which depends on the value of the secret bit `k[i]` at position `i`. Other channels include the termination behavior of the program or the resource exhaustion which, if dependent on high data, may result in secret information leakage.

The transparency offered by high level programming languages hides many implementation details which may be exploited by attackers with knowledge of low level details such as caches, pipelines, CPU models or even power consumption [15]. Language-based techniques often operate based on a semantics of the programming language, which ignores such implementation details. As a result, a program which is proved secure at the source code level may by insecure with respect to attackers that observe features not covered by the programming language semantics. Consider for instance the program in Fig. 1.9, where `sec, sec1, sec2` are high security variables and `pub, pub1, pub2` are low security variables. The program would be

considered secure with respect to an attacker who has access to the final value of low variables and can count the number of assignments performed at the source code level. In fact, the program never assigns to low variables and always executes the same number of instructions. However, depending on the truth value of the

```
if (sec) {sec1 := pub1;}
else     {sec2 := pub2;}
pub := pub1;
```

Figure 1.9: Cache leakage

boolean variable `sec`, the execution time of this program may vary. Indeed, if `sec` is true, the last assignment can take less time as the value of `pub1` is already in the data cache. If `sec` is false, the program may have to load both `pub1` and `pub2` from the memory, which takes longer time. Similar examples may use instruction caches, pipelines or other architecture dependent details. A possible solution to this issue is to explicitly model all these implementation details at the language semantics level, which comes at the price of a much harder verification process.

More complex computational models include concurrent and distributed systems, which give rise to additional information flow channels. The nondeterminism inherent in these models can be exploited by attackers in several ways to leak sensitive information. For instance, in a multithreaded setting, the timing behavior may affect, through the scheduler, the execution order of low events and introduce *internal* timing channels [193]. Consider the multithreaded program in Fig. 1.10 where `l` and `h`, respectively, denote low and high shared variables, `||` denotes the parallel composition and `delay(t)` delays execution of the program for the amount of time specified by `t`. Both threads are secure in isolation. However, under reason-

```
if (h) {delay(100);}
else   {delay(1);}          ||     l:=2
l := 1;
```

Figure 1.10: Internal timing leakage

able schedulers, the assignment `l:=1` will execute last if the secret `h` is true. Similar channels can be encoded into the stochastic behavior of the system and are known as *probabilistic* channels.

The very nature of concurrency requires reactive/interactive models. In a classical client-server communication scheme, an untrusted client may exchange several messages with the server and the sequence of such messages can encode sensitive information. Consequently, the security condition must cater for more fine-grained

channels concerning occurrences/non-occurrences of sensitive events or even the way low events are interleaved with high events. The simple program, $\texttt{in}_H\texttt{(x);out}_L\texttt{(1)}$, which inputs a value on a high channel and always outputs `1` on a low channel, can leak sensitive information. Indeed, the event on low channel signals that some message was input on high channel. This can be sufficient for an attacker to disclose sensitive information in some contexts, for instance whether a user visits a medical web site. The program in Fig. 1.11 presents an information channel through the observation of the sequence of low events. The program reads a secret number,

```
in(H, secret);
i:=0; max := Max;
while (i<= max) {
if (i == secret) out(L1, "Found");
else          out(L2, "Trying...");
i++;
}
```

Figure 1.11: Trace leakage

known to be a non-negative integer in the range `0` to `Max`, from high channel `H`, and loops `Max+1` iterations outputting the string `Trying...` on low channel `L2` for `Max` times and the string `Found` on low channel `L1` once. An attacker who observes the outputs on low channels synchronously can reveal the entire password by counting the number of `Trying...` messages received on channel `L2` prior to receiving the messages `Found` on channel `L1`. However, if outputs are not necessarily observed in the order they are produced, the attacker can not in general establish such a relation, hence the program may in some contexts be considered secure.

Information flow conditions for reactive/interactive systems can sometimes be expressed over streams of inputs and outputs [57, 182]. The system can be interpreted as a (nondeterministic) transformer between input streams and output streams, and security is then defined as a property of the transformer over the streams. When possible, this allows a sort of reduction to relational, i.e. initial state-final state, noninterference, as program inputs can be read upfront and program outputs can be produced upon termination. For deterministic programs this is indeed the case, as shown in [80]. However, programs that make nondeterministic choices and expose these choices to high users can leak information through user *strategies*.

The program in Fig. 1.12, from [224], nondeterministically chooses `0` or `1` and sends the value to a high user on channel `H2`. The high user inputs a value on channel `H1` and the `XOR` of the two values is sent to a low user on channel `L`. The low user can observe either `0` or `1`, independently of the high value input on `H1`, hence the program seems secure. Suppose now the high user is a spy who wants

```
x:= 0||1;
out(H2, x);
in(H1, y);
out(L, (x XOR y));
```

Figure 1.12: Leakage Through User Strategies

to transmit a secret bit `z` to the low user. The spy can then input `(z XOR x)` on channel `H1` and, from the identity `(x XOR z XOR x ) = z`, the low user will receive the exact value of the secret `z`.

Which information flow channels are a security concern depends on the particular context and on the power of the attacker. The existing security conditions can be categorized with respect to three parameters: the computational model (batch job, reactive, interactive), the attackers' power (initial-final state, traces, timing, termination) and the sensitive information they protect (initial high state, occurrences of high events, sequences of high events). In this thesis, we study policies and techniques that apply to several of the information flow channels described in this subsection.

1.2.3 Information Release Policies

The primary goal of a computing system is to offer a range of functionalities and features to the users to perform certain tasks. However, functionality and security in general can be two conflicting requirements. The more functionality a system provides, the less information is maintained secure. The complete separation between high and low computation assumed by the noninterference condition is not always met by practical applications. For instance, a simple authentication routine that implements a password checking program reveals whether the input password (low) equals the correct password (high) and thus the noninterference condition fails. Similarly, for integrity, an untrusted input can safely be considered trustworthy after a *sanitize* function has been applied. Again, the noninterference condition is violated due to the flow of information from untrusted input to some trusted output. As a result, controlled release of secret information and controlled upgrade of untrusted information, is a crucial requirement for information flow control to be useful [202, 26]. In the security literature, this deliberate release of secret information is known as *declassification*, and, dually, for integrity, as *endorsement*. In the examples above, one would like to declassify the password checking result and to endorse the sanitized input. Information *erasure* is another related notion which can be described as an increase of information security by erasing the sensitive data [76]. For instance, a web shopping application must erase the users' credit card numbers once the transaction has gone through.

All these notions are necessary to model a common requirement of information

security, namely that the security of information changes with time. As an example, consider the information system of any organization that involves multiple users with different security clearances, for instance an e-government online system. The system should provide access to public documents to all citizens accessing the system. When a document is classified as public, this information should be released to the citizens, which is a declassification requirement. Then, when a citizen starts working for the government, she becomes a registered user and, depending on the department she is assigned to, more information is made available. The security of the information can increase or decrease over time. For instance, if the employee gets promoted or moves to another department she should get access to new information and lose access to the old. Similarly, if the employee leaves the system as a result of getting fired, it should not be possible to access the internal information. The dynamic nature of security policies has been recognized by different researchers as a crucial property an enforcement mechanism should take into account. Existing security models study different aspects of information release including *what* information is released, *who* performs the information release, *where* the information is released and *when* the information release can take place [202]. A unified framework that embodies all these dimensions has been considered an open issue, and a solution is proposed in this thesis.

1.2.4 Enforcement: Static vs. Dynamic

The literature has two main approaches to information flow control: *static* enforcement and *dynamic* enforcement. From the beginning information flow research has been "riding the roller coaster" between static and dynamic mechanisms [199]. The static analysis approach is appealing as it allows to verify and certify the information security requirements at compile time, and thus avoids the runtime overhead. Security type systems are by no means the most used approach for static analysis. They mainly impose Denning's approach [100] by assigning security labels to program data, e.g. variables, fields, and enforcing separation between high and low computation, essentially by maintaining the invariant that no low computation occurs in a high context. To get a flavor of how a security type system works, consider an excerpt of typing rules from [197], as shown in Fig. 1.13.

$$\frac{\vdash exp : public}{[public] \vdash l := exp} \qquad \frac{\vdash exp : public \quad [public] \vdash C_1 \quad [public] \vdash C_2}{[public] \vdash \texttt{if } exp \texttt{ then } C_1 \texttt{ else } C_2}$$

Figure 1.13: Security Type System

Suppose each program variable is assigned a security label according to the security lattice in Fig. 1.4a. As a result, the security type system must prevent flows of information from secret variables to public variables. The rule on the left

considers typing of an assignment statement. Basically, it says that an assignment to a public variable (we assume l is public) is allowed only if all variables in *exp* are public. The judgment $[public] \vdash C$ means that the program C is typable in the security context *public*. The security context is mainly needed to track implicit flows. The rule on the right says that a conditional statement is typable in a *public* security context only if the branch condition *exp* and the sub commands C_1, C_2 are typable in a *public* security context. For instance, if variable h has security type *secret*, the type system would reject the program `if` h `then` $l := 0$ `else` $l := 1$ since h can not be typed in a *public* context. Similarly, the explicit flow in $l := h$ is prevented by the first rule. Security type systems are desirable due to their simplicity and the efficiency of type checking. However, many systems have complicated security policies which can not always be enforced by type systems. For example, the secure program `if` h `then` $l := 1$ `else` $l := 1$ would incorrectly be ruled out by the type system above.

Dynamic techniques make use of the program runtime information to perform information flow analysis. Another program, often called a security monitor, supervises the execution of the target program and checks at runtime that no security policy violation occurs. Broadly, the monitor enforces the invariant that no assignment from high to low variables occurs either explicitly or implicitly through program control structures. If a violation occurs the monitor can take several countermeasures, for instance it can decide to terminate the execution of the program [153]. Dynamic enforcement of information flow is particularly useful for highly dynamic languages, typically used on the web, for instance Javascript, where the content is often unknown until runtime. Besides the runtime overhead, dynamic monitoring can not always enforce noninterference policies as it is well known that noninterference is a hyperproperty, and thus it can not be enforced by looking at one execution at a time [164]. As a result, dynamic enforcement techniques typically rely on static processing to increase precision, as we discuss later [153].

1.3 State of the Art and Beyond

In this section we give an overview of the state of the art in information flow security and define some research problems. We start with a quick historical background of the seminal ideas that led to the IFC as a research area, then we discuss recent challenges that the community has gone through and finally conclude with our research problems giving a taste of the contributions made by this thesis to solve them.

1.3.1 Historical Background

The realization of the importance of formal security models dates back to the early seventies when several research projects in this area were funded by the US Department of Defense. Problems with providing strong security guarantees, both at the

design and the implementation level, have led to the need to develop new mathematical methods to prove that the design satisfies predefined security requirements and that the subsequent implementation faithfully conforms to the design.

Noteworthy, the model developed by Bell and La Padula [55] in 1973 aimed at providing a formal basis for confidentiality using access control policies. In a nutshell, subjects and objects are assigned to *security classes* which form a hierarchy of *security levels*. This gives rise to a *multilevel* security requirement, which essentially ensures that a subject at a higher level does not convey information to a subject at a lower level. The requirement is formalized in terms of the *No read up* policy, stating that a subject can only read an object of less or equal security level, and the *No write down* policy, stating that a subject can only write an object of greater or equal security level. Moreover, the model can be enriched with some form of discretionary access control where a subject can grant permissions to another subject to access some object. The Bell-La Padula model has influenced the design and implementation of the first security-aware operation systems, such as Multics [222]. However, the model is known to have several limitations, above all the presence of information flow channels and the difficulty to cope with integrity requirements [209]. To overcome some of these limitations, other formal security models have been proposed, including the Biba [56] and Clark-Wilson [82] integrity models, and the Chinese Wall model [59], which incorporates both confidentiality and integrity.

The work by Denning and Denning [100] can be considered as the successor of the Bell-La Padula model for information flow security. The authors introduce a lattice of security levels for policy specification and, at the same time, observe that static program analysis can be a good solution to the confinement problem introduced earlier by Lampson [151]. For dynamic information flow, the work by Fenton [113] is arguably considered the seminal contribution in the area. Fenton describes an abstract machine enriched with security labels, called data marks, which decorate the storage locations and the program counter in order to prevent illegal information flows.

Semantic models of information flow have been developed in parallel with the static and dynamic enforcement mechanisms mentioned above. Informal attempts to information flow models start with the *confinement* problem [151], which defines covert channels as mechanisms for unintentional transfer of confidential information in computer programs. In fact, confinement requires that systems do not leak confidential data, even partially. This idea was later formalized by Cohen [86, 87], who introduced the notions of *strong* and *selective* dependency, quite close to what today is known as noninterference and declassification. Noninterference, introduced by Goguen and Meseguer [122], formally defines the intuition that one group of users does not *noninterfere* with another group of users, if what the former group does has no effect on what the latter can see. Noninterference can safely be considered as the most studied semantic security condition in information flow security.

Both lines of work, the semantic conditions and the enforcement mechanisms, were significant steps towards formalizing a secure system as defined in Sect. 1.2.

What was missing was a formal justification, i.e. a soundness argument, that would relate the two and thus provably show that the enforcement mechanism indeed ensures the semantic security condition. This relation was given by Volpano and Smith [218] who showed that the security type systems guarantee the noninterference security condition. Later works, this thesis included, elaborate on these seminal ideas and attempt to push the boundary in terms of theoretical foundations, verification techniques and practical tools with information flow guarantees.

1.3.2 Recent Developments

Information flow control, and its branch of language-based security, is now a well-established research area. Over the past four decades, a vast number of methods and techniques have been developed to specify and verify the end-to-end security requirements provided by information flow control. Although everyone seems to agree on the usefulness of information flow policies, there still exists a debate about its practical adoption in production systems. Living at the confluence of programming languages and security, IFC inherits, in addition to its own challenges, also known issues and limitations from both areas. On the other hand, information disclosures, Heartbleed being the last of a large and growing list, appear frequently in today's software and current security solutions are all but satisfactory. In short, information flow control is definitely a pressing problem we should work on and find new better solutions.

Remarkable progress has been made in advancing the state of the art in terms of theoretical foundations and practical enforcement. Here we quickly survey on the main challenges addressed by researchers over the past years.

Relaxing Secure Information. The noninterference condition, as pointed out earlier, is not well suited for some systems. As a result, a lot of research effort has been dedicated to the modeling of controlled release of secret information. Noteworthy, several notions such as gradual release [28], admissibility [120], abstract noninterference [119], delimited release [198], trusted declassification [134], noninterference until [75], relaxed noninterference [156] and many others, have been introduced to account for allowed flows of information. A less studied notion, information flow integrity, has also addressed issues of controlled information upgrade, known as information endorsement [176, 40, 26]. Other related notions include information erasure and more generally dynamic security policies, which have been considered in [76, 60, 24, 35]. A recent survey proposes a classification of different approaches to information release [202].

Expressiveness and Concurrency. The nondeterministic and probabilistic nature of concurrent and distributed computations gives rise to information flow channels which are otherwise ignored by the noninterference condition. For instance, the information channels described in Fig. 1.10-1.12 are typical of these models. Consequently, alternative security conditions have been proposed to cope with more the complex channels arising in these execution contexts. In the literature, these are known as possibilistic security conditions. For instance, nondeducibility

on strategies [224] can be used to rule out covert channels similar to the one in Fig. 1.12. Frameworks that aim at unifying the possibilistic security conditions have also been proposed, including the selective interleaving functions by McLean [164], the modular assembly kit by Mantel [159] and process algebra classifications by Focardi and Gorrieri [114]. Other security models which address information flow for concurrent and multithreaded programs are the PER model by Sabelfeld and Sands [200], the equational security condition by Leino and Joshi [142], low determinism [192, 166], and several bisimulation-like conditions [207, 91, 58, 200]. Various attempts have been made to express these conditions using logics of knowledge [128, 125, 35], mu-calculus [138, 169] or other non conventional logics tailored to information flow properties [19, 104, 84].

Attack Models. The pervasive nature of information flow channels enables different attacker models. The observation power of these attackers determines the type of information flow channels to protect against. For instance, the capability of an attacker to observe program (non)termination has given rise to notions of termination *sensitive* and termination *insensitive* attacker models [25, 144]. Similarly, an attacker can exploit the timing behavior of the program which may depend on a secret and thus reveal information [230, 131, 16]. Another line of work, known as *quantitative* information flow, studies information-theoretic bounds of the secret information released by an application [81]. As opposed to this, *qualitative* information flow focuses on properties of secret information. Also, complexity-theoretic approaches have been proposed to reason about polynomial time attackers using computational notions of indistinguishability [152] or probabilistic programming languages [186, 77].

Security Labeling. An important prerequisite of applying information flow control is the security classification or labeling of information sources and sinks. In the simplest setting, this labeling is taken for a two point security lattice as in Fig. 1.4, where information originating from high sources is disallowed to flow to low sinks. For example, a game application reading the list of phone contacts, labeled as high, should be disallowed to send this information to an untrusted server, labeled as low. However, in some cases the security labeling can be challenging to define, in particular for low level code [36]. More complex applications, such a distributed and concurrent programs, may involve principals with different security requirements which mutually distrust one another. Consequently, the security lattice needs to be finer to reflect the security relationships between each pair of principles. Noteworthy, Myers and Liskov [175] introduced the decentralized labeled model (DLM) which allows to express such security policies, also in the presence of declassification, for mutually distrusting principles that can explicitly transfer ownership. Several authors have studied security policies using DLM, including [74, 60, 41].

IFC Integration. Information flow control alone would not be sufficient to provide security in an end-to-end fashion. The main reason is that system-wide security crosses the boundaries of single applications and requires interaction with other systems whose security is provided by other means, such as encryption or

access control. Hence, it becomes vital to integrate information flow techniques and other security techniques in a secure manner. An important line of work addresses the problem of secure composition by integrating encryption, access control or other security mechanisms in a unified framework [43, 177, 28, 216, 115].

IFC Enforcement. The actual enforcement of information flow policies is probably the Achilles' heel in information flow research. Clearly, one can come up with fancy semantic security conditions able to express all sorts of security policies; however, this would be of limited use if an enforcement mechanism can not accept or reject applications that satisfy or violate these policies, respectively. In fact, one of the drawbacks of current enforcement mechanisms are the constraints they impose on the way programs are written, therefore putting the burden on the programmer and making their use limited. Security type systems have dominated the static verification approaches to information flow [218, 197]. For systems with complicated security policies researchers have proposed more precise verification methods including flow-sensitive security types [139], dependent types[177], abstract interpretation [119, 149], relational logics [52, 51, 20], model checking and symbolic execution [37, 178, 104, 36] or theorem proving [97]. Which verification method to use, is application and policy dependent and requires to find the right trade-off between verification effort and policy expressiveness.

Dynamic information flow analysis has been successfully applied to web security, where code and data are not always known before runtime. Much progress has been made to improve the precision of dynamic analysis and thus be able to accept more secure programs [153, 30, 130]. Recent work discusses the trade offs with static analysis and shows that, in some cases, dynamic approaches can be as precise as security type systems [199, 194]. Being a hyperproperty [85], noninterference can not always be enforced by monitoring one execution at a time [153]. To overcome this limitation, researchers have proposed combinations of dynamic and static analysis, known as *hybrid* analysis [153, 29, 79, 78]. More recently, a technique known as *secure multi-execution*, has been introduced to enforce dynamically noninterference in a language independent way. The core idea of secure multi-execution is to execute as many copies of the program as the number of security levels and ensure that the lower security copies are run before the higher ones, in a carefully synchronized manner [101].

It is worth noting that a significant line of work uses *taint* tracking to test programs for confidentiality and integrity bugs. Broadly, untrusted sources can be tagged as tainted and, by propagating the taint value during program execution, one can ensure that no tainted value can modify a trusted sink. The big advantage of this technique is scalability. In fact, it has been successfully applied to check for security bugs in low level code [180, 179] and Smartphone apps [110, 94]. However, taint tracking sacrifices soundness for scalability and it can not directly be used for verification; for instance, implicit flows can not be handled by taint analysis.

Pervasive IFC. Information flow control, both static and dynamic, has been extensively applied to the entire software stack, including the application level [197], the systems level [173, 229, 108] or lower levels such as bytecode [37, 53]

machine code [36, 49] and hardware [32, 214]. Noteworthy, recent developments in virtualization technologies advocate thin software layers, for instance hypervisors and separation kernels, which allow different users and systems to share hardware resources [232]. Being relatively small-sized, provable information flow security of these layers is feasible, as shown by several projects, for example the seL4 OS kernel [147] or the PROSPER separation kernel [92, 36].

IFC Tools. Several practical tools for programming with information flow control have been implemented for the mainstream languages. The JIF compiler developed at Cornell was the very first tool to provide information flow support for JAVA programs. JIF uses security type systems to statically enforce information flow policies expressed in the DLM model [7]. Flow Caml is an extension of the Objective Caml language with a type system enforcing information flow policies [1]. Other tools with information flow support include Paragon [11] which applies security type systems, JOANA which uses program dependence graphs [8] and ENCoVer which is based on dynamic symbolic execution [39], all targeting JAVA programs. On the dynamic enforcement side, most tools target web languages, mainly JavaScript. In particular, JSFlow [9] and FlowSafe [2] perform dynamic information flow tracking for Javascript code in the browser, while FlowFox [6] is based on the secure multi-execution approach.

1.3.3 Research Problems and Results at a Glance

In this subsection we discuss several research problems in the area of language-based information flow security with particular emphasis to those tackled in the rest of this thesis. Over the past years, the need for strong fine-grained semantic-based security guarantees provided by IFC has been contrasted by the difficult integration with existing security practices and infrastructures, which, together with a demanding verification process, has mainly received the attention of researchers, and has yet to be fully adopted by the industry.

The underlying reason of this limited adaption is simple: reasoning about security requirements in an end-to-end fashion is a *hard* problem. Besides the fundamental limitations of undecidability and computational complexity, information flow policies concern properties over sets of executions, i.e. hyperproperties, and traditional reasoning and verification techniques can not be applied *as is*. Devising semantic frameworks which are expressive enough to characterize the desired policies, yet amenable to efficient verification, is an open issue. Moreover, as mentioned previously, security requirements are subject to dynamic changes and a controlled release of secret data is necessary at different points of the program execution. To address this problem, in the second chapter we propose a logic, called *temporal epistemic logic* which allows to reason about noninterference and declassification and, at the same time, to unify the different dimensions of information release in a single framework. Reasoning about information flow policies in terms of the attacker's knowledge is simple and intuitive, and it provides a precise syntactical characterization of the security policy, which can be verified using the techniques developed in

this thesis. Intuitively, the insecure program `out(pwd)` can be ruled out as follows: prior to program execution the attacker's *knowledge* is that the password `pwd` can be any string. After the execution, knowing the program code, the attacker can narrow down his uncertainty and *learn* that the password is exactly the string that he observes. As a result, this increase of knowledge, i.e. decrease of uncertainty, can be used to consider the program insecure.

Language-based security has been dominated by the state-based notions of information flow. However, more abstract information flow conditions have been proposed for other computational models, including trace-based conditions and process algebras, or more traditional interactive and reactive programming languages. These conditions rely on properties of their target systems and they are specialized to solve technical problems such as compositionality, asynchrony and verification. As a result, they turn out to be overly complicated and it is often unclear what security property they actually enforce. Reasoning about knowledge is quite useful for characterizing the security of these systems as well. In the third chapter, we propose a knowledge-based framework for reasoning about trace-based information flow conditions. The security model brings out what events on channels an observer can see and what observations on events are to be protected. This allows a very general treatment of secret information and declassification, both as high level input and output events, and as relationships between events, say ordering, multiplicity, and interleaving. Again, a syntactical characterization using temporal epistemic logic has been studied.

Another challenge in information flow security is the interplay between confidentiality and integrity, which may give rise to security violations similar to Heartbleed. We address this problem in the fourth chapter where the goal is to enforce confidentiality in presence of *active* attacks. This notion is known as *robustness* and it requires that an active attacker, who can insert code at certain program points, is unable to learn more information than a passive attacker, who can merely observe the public state of the program. The problem is realistic in different settings such as secure program partitioning [228], where a possibly corrupted server can compromise the security of the entire system, or in languages as Javascript, where possibly untrusted code can be received over the network and integrated in the system, for instance by means of the tricky *eval* construct [181]. We address this problem in the context of the abstract interpretation framework [89], using the weakest precondition calculus. Among other results, we devise sufficient conditions to ensure robustness with respect to active attacks.

In the second part of the thesis, we return to the pressing problem of enforcement of information flow policies and propose different algorithms for precise static verification of trace-based information flow conditions. Precise verification of information flow over program traces is challenging since in general trace properties do not compose, thus a global analysis of the system may be required in the worst case. Several logics for information flow properties have been proposed. Rarely, however, have they been automated for mainstream programming languages as we do in this thesis. In particular, we leverage the recent advances in automated theorem

proving and propose symbolic verification algorithms to combat the state explosion problem [141]. As expected, our methods may still suffer from scalability issues, however they are able to analyze programs with complex information flows which appear in many real applications. We show by means of several case studies that programs taken from practical scenarios are well within the reach of our tools.

In the fifth chapter we use *symbolic execution* to extract a model of the program runtime behavior and subsequently verify it against the target security policies, expressed in epistemic logic, by means of an *epistemic model checker*. Alternatively, the model checking problem is transformed into a first order logic formula, which only contains existential quantifiers, thereby an *SMT solver* is used to perform the verification efficiently. In the sixth chapter we propose a novel approach to relational verification of information flow for low level code. Low level verification is quite challenging due to the highly optimized nature of the code and the absence of structured control flow. In addition, low level features such as symbolic jumps, symbolic memory or usual side effects can significantly complicate the verification process. Again, we approach this problem by using symbolic execution techniques to propagate the relational verification conditions and then discharge them using first-order reasoning.

Another challenge in information flow research is the integration of language-based security techniques with realistic programming languages, together with practical tool support that can be easily used by the programmers. This thesis puts a step forward also in this direction. We have implemented a tool prototype, ENCoVer, as an extension of Java PathFinder [184], a software model checker for Java bytecode developed at NASA. ENCoVer performs information flow analysis of Java programs in a *push-button* fashion and allows the programmer to externally specify the security requirements, thus separating the program text from the policy. Several case studies show the capabilities and the limitations of ENCoVer. Similarly, we have implemented an automated tool that integrates with SMT solvers to automate the verification task of machine code. The tool transforms ARMv7 binaries [23] into an intermediate, architecture-independent format extending the BAP toolset [62]. The tool has been successfully used to show the absence of information flow channels on a separation kernel system call handler, which mixes hand-written assembly with gcc-optimized output, a UART device driver and a crypto service modular exponentiation routine [36].

1.4 Thesis Results

In this section we give a short overview of the papers included in this thesis, together with a statement of the author's contributions. We start with a gentle description of the basic concepts needed to understand the thesis results, and then go on to describe the actual results. The thesis consists of five papers, some of which have been minimally revised to improve the overall presentation and to include additional proofs.

1.4.1 A Simple Worked-Out Formalization

In this subsection we elucidate the overall specification and verification approach by means of a formalization and a simple running example. To this end, we first introduce a fragment of an imperative language, including syntax and operational semantics, and show how the language is used to reason about the noninterference condition. Then we conclude with a logic and a symbolic verification method which are used to specify and verify noninterference, respectively. All these steps are accompanied by the running example.

Computational Model. We study a simple imperative language extended with a synchronous output statement that, over the course of a computation, causes information to be leaked to an observer. Besides the output statement "$\texttt{out}(e)$", the features of the language are commonplace: assignments, conditionals, a primitive data type of values belonging to a finite set *Val*. Loops are excluded to avoid clutter. The grammar of the language is given in Fig. 1.14. Programs are ranged over by P, identifiers by x, values by v, and expressions by e.

$$\begin{aligned} P ::= {} & \texttt{skip} \mid \texttt{out}(e) \mid x{:=}e \mid P \texttt{ ; } P \\ & \mid \texttt{if } e \texttt{ then } P \texttt{ else } P \end{aligned}$$

Figure 1.14: A Simple Imperative Language

$$\frac{(e,\sigma) \longrightarrow v \in \mathit{Val}}{(\texttt{out}(e),\sigma) \xrightarrow{v} (\texttt{skip},\sigma)} \qquad \frac{(e,\sigma) \longrightarrow v \in \mathit{Val}}{(x := e,\sigma) \rightarrow (\texttt{skip},\sigma[x \mapsto v])}$$

$$\frac{(P_0,\sigma) \xrightarrow{(\alpha)} (P_0',\sigma')}{(P_0; P_1,\sigma) \xrightarrow{(\alpha)} (P_0'; P_1,\sigma')} \qquad \frac{}{(\texttt{skip}; P_1,\sigma) \rightarrow (P_1,\sigma)}$$

$$\frac{(b,\sigma) \longrightarrow \mathit{true}}{(\texttt{if } b \texttt{ then } P_0 \texttt{ else } P_1,\sigma) \rightarrow (P_0,\sigma)} \qquad \frac{(b,\sigma) \longrightarrow \mathit{false}}{(\texttt{if } b \texttt{ then } P_0 \texttt{ else } P_1,\sigma) \rightarrow (P_1,\sigma)}$$

Figure 1.15: Small Step Operational Semantics

A store is a finite map $\sigma : x \mapsto v$, and $\sigma(e)$ is the value of e in store σ. An execution state is a pair (P,σ). The execution of a program generates observable actions (or events) belonging to *Act* and ranged over by α ($\mathit{Act} = \mathit{Val}$). The

transition relation $(P,\sigma) \xrightarrow{\alpha} (P',\sigma')$, or $(P,\sigma) \rightarrow (P',\sigma')$, states that by taking one execution step in the execution state (P,σ) the execution generates the visible event α, if it is present, and the new execution state is (P',σ'). We write $(P,\sigma) \xrightarrow{(\alpha)} (P',\sigma')$ where α is optional. The small step operational semantics of the language is shown in Fig. 1.15. An *execution* is a finite sequence of execution states.

$$\pi = (P_0,\sigma_0) \xrightarrow{(\alpha_0)} \cdots \xrightarrow{(\alpha_{n-1})} (P_n,\sigma_n) \tag{1.1}$$

We write $len(\pi)$ for the length (number of transitions) of the execution π. An *execution point*, or simply *point*, is a pair (π,i) where $0 \leq i \leq len(\pi)$. We write $\sigma(\pi,i)$ or σ_i for the store at point (π,i). The power of the attacker is modeled by providing a function *trace* mapping execution points to traces that represent what the attacker has been able to observe so far. A trace τ is a sequence of actions. We use the function *trace* such that $trace(\pi,i)$ is the sequence of events α_j such that $0 \leq j < i$ and α_j exists. We write $trace(\pi)$ for $trace(\pi, len(\pi))$. For instance, the trace of the execution (1.1) is: $(\alpha_0)(\alpha_1)\cdots(\alpha_{n-1})$. This definition corresponds to the so called perfect recall attacker [112], i.e. having memory of past observations. A (program) model $\mathcal{M}$ is then defined as the set of all executions originating from some designated set of initial states, for instance of the shape (P_0,σ_0) where P_0 is a fixed initial program.

Example 1.4.1 (Running Example) *Program P consists of two conditional statements over a boolean identifier h and it always outputs 0 or 1.*

$$P ::= \left[\begin{array}{l} \texttt{if } h \texttt{ then out}(0) \texttt{ else skip;} \\ \texttt{if } \neg h \texttt{ then out}(1) \texttt{ else skip} \end{array}\right.$$

The program model $\mathcal{M}$ contains exactly two executions, π_0 and π_1, starting with initial stores $\sigma_0(h) = true$ and $\sigma_1(h) = false$, respectively. Therefore, $trace(\pi_0) = 0$ and $trace(\pi_1) = 1$.

Knowledge and Security. We can use the program models as defined above to reason about information flow properties using the notion of knowledge. In this discussion we focus on noninterference which we have already stated informally. Noninterference can be defined on a two-level security lattice (recall Fig. 1.4) and it requires that no information about initial values of high identifiers can escape the program through the output statement. Again, to avoid clutter, we assume the store consists of high identifiers only. For instance, the identifier h in the running example is labeled as high security.

Following the recipe in Fig. 1.3, we define the various ingredients as follows. The system model is the program model $\mathcal{M}$, which is public knowledge. The security policy and the attacker model consist of the pair $\mathcal{A} = (\mathcal{O},\mathcal{P})$, where $\mathcal{O}$ defines the attacker's power and $\mathcal{P}$ defines what information to protect. In our setting, $\mathcal{O}$ is the *trace* function and $\mathcal{P}$ is the set of program identifiers (since we assumed they

are all labeled as high). The knowledge of an attacker who knows the program model $\mathcal{M}$ and observes the output actions is then defined with respect to a point (π, i) as follows.

$$\mathcal{K}(\pi, i, \mathcal{A}) = \{\sigma(\pi', 0) \mid \pi' \in \mathcal{M} \wedge \mathit{trace}(\pi, i) = \mathit{trace}(\pi', i')\} \tag{1.2}$$

where $0 \leq i' \leq \mathit{len}(\pi')$. Intuitively, $\mathcal{K}(\pi, i, \mathcal{A})$ represents the set of initial (secret) stores that the attacker holds for possible based on its observations up to point (π, i). Namely, an attacker who observes $\mathit{trace}(\pi, i)$ can not tell apart which of the executions starting from an initial store in $\mathcal{K}(\pi, i, \mathcal{A})$ is the one that is actually executing. The security condition ensures that for each execution point $(\pi, i+1)$, the attacker's knowledge is not greater[3] than its knowledge at the previous point (π, i).

$$\text{For all } \pi \in \mathcal{M},\ \ \mathcal{K}(\pi, i, \mathcal{A}) \subseteq \mathcal{K}(\pi, i+1, \mathcal{A}),\ \text{where}\ \ 0 \leq i < \mathit{len}(\pi) \tag{1.3}$$

We exercise the above security condition on the running example.

Example 1.4.2 *Let* $P_1 ::= \texttt{if}\ \neg h\ \texttt{then}\ \texttt{out}(1)\ \texttt{else}\ \texttt{skip}$. *Then the program text in Example 1.4.1 has the following model (using the rules in Fig. 1.15).*

$$\mathcal{M} ::= \begin{cases} \pi_0 = (P, \sigma_0) \rightarrow (\texttt{out}(0); P_1, \sigma_0) \xrightarrow{0} (\texttt{skip}; P_1, \sigma_0) \rightarrow (P_1, \sigma_0) \rightarrow (\texttt{skip}, \sigma_0) \\ \pi_1 = (P, \sigma_1) \rightarrow (\mathit{skip}; P_1, \sigma_1) \rightarrow (P_1, \sigma_1) \rightarrow (\texttt{out}(1), \sigma_1) \xrightarrow{1} (\texttt{skip}, \sigma_1) \end{cases}$$

By definition (1.2), we can compute $\mathcal{K}(\pi_0, 2, \mathcal{A}) = \{\sigma_0, \sigma_1\}$ *and* $\mathcal{K}(\pi_0, 3, \mathcal{A}) = \{\sigma_0\}$. *As a result, the security condition (1.3) is violated, i.e.* $\mathcal{K}(\pi_0, 2, \mathcal{A}) \not\subseteq \mathcal{K}(\pi_0, 3, \mathcal{A})$. *Indeed, after the attacker observes the output value* 0*, she can refine her knowledge and learn that* σ_1 *is not a possible initial store. Hence, the program is deemed insecure.*

A Logic for Information Flow. We now propose to use a fragment of temporal epistemic logic to express the security condition in (1.3) in a syntactical manner. The language of formulas ϕ, ψ in temporal epistemic logic is given as follows:

$$\phi, \psi ::= \mathit{init}_x(e) \mid \phi \wedge \psi \mid \neg\phi \mid K\phi \mid G\phi$$

The language contains atomic propositions $\mathit{init}_x(e)$ expressing that the value x in the initial state is identical to the value of e in the current state. The purpose of the initial state predicate $\mathit{init}_x(e)$ is to capture what is known "now" of the initial store. The operator K is the epistemic knowledge operator. $K\phi$ holds if ϕ holds in any state equivalent to the current state. The "always" operator $G\phi$ meaning that ϕ holds in any future state. Various connectives are definable in the language including standard derived boolean operators such as implication $\rightarrow$, universal quantification over the finite set of values $\forall x.\, \phi = \bigwedge_{v \in \mathit{Val}} \phi[v/x]$ or the

[3]Please recall that the smaller the set $\mathcal{K}$, the greater is the attacker's knowledge.

epistemic possibility operator $L\phi = \neg K(\neg\phi)$ meaning that ϕ holds for at least one epistemically equivalent state. We use these as syntactic sugar. Satisfaction relative to program model $\mathcal{M}$ is defined as $\mathcal{M} \models \phi$ iff for all $\pi \in \mathcal{M}$, $(\pi, 0) \models \phi$. Here we report the formal definition of the satisfaction relation at an execution point (π, i) for selected logic operators.

$$
\begin{array}{ll}
(\pi, i) \models \mathit{init}_x(e) & \text{iff } \sigma(\pi, 0)(x) = \sigma(\pi, i)(e) \\
(\pi, i) \models G\phi & \text{iff for all } i' \geq i,\ (\pi, i') \models \phi \\
(\pi, i) \models K\phi & \text{iff for all } (\pi', i') \text{ s.t. } \mathit{trace}(\pi, i) = \mathit{trace}(\pi', i'),\ (\pi', i') \models \phi \\
(\pi, i) \models L\phi & \text{iff there exists } (\pi', i') \text{ s.t. } \mathit{trace}(\pi, i) = \mathit{trace}(\pi', i'),\ (\pi', i') \models \phi
\end{array}
$$

Let $h_1, \cdots, h_n$ be the set of program identifiers (all labeled as high security). Then we can express the security condition (1.3) in the logic.

$$\mathcal{M} \models\ \forall v_1, \cdots, \forall v_n\ G\ L(\mathit{init}_{h_1}(v_1) \wedge \cdots \wedge \mathit{init}_{h_n}(v_n)) \tag{1.4}$$

By definition of the satisfaction relation, the formula (1.4) can be interpreted as follows. Consider the execution point $(\pi, 0)$ for some $\pi \in \mathcal{M}$. Then, along any execution point (π, i) of execution π (cf. G operator), there exists an execution point (π', i') with the same trace as (π, i) (cf. L operator), which can originate from any value initially assigned to the program identifiers (cf. $(\mathit{init}_{h_1}(v_1) \wedge \cdots \wedge \mathit{init}_{h_n}(v_n))$ condition). The following example illustrates this point.

Example 1.4.3 *We apply the logical characterization (1.4) to the model in Example 1.4.2. This requires to check whether $\mathcal{M} \models \forall v\ G\ L(\mathit{init}_h(v))$ holds. Namely,*

$$\mathcal{M} \models\ G\ L(\mathit{init}_h(\mathsf{true}))\ \wedge\ G\ L(\mathit{init}_h(\mathsf{false}))$$

In particular, it must be the case that $(\pi_0, 3) \models\ L(\mathit{init}_h(\mathsf{false}))$. From Example 1.4.2, π_1 is the only execution where $\mathit{init}_h(\mathsf{false})$ holds and $\mathit{trace}(\pi_1) = 1$. However, $\mathit{trace}(\pi_0, 3) = 0$, hence a contradiction. As expected, the formula is not true in the model.

A Symbolic Verification Method. We have seen how reasoning about knowledge can be used to capture the noninterference condition and how the reasoning can be expressed syntactically using temporal epistemic logic. Now we address the verification problem and present a symbolic method to check whether an epistemic formula of the shape as in (1.4) holds of a program as presented in Fig. 1.14.

The main idea is to reason about the behavior of a program by means of forward symbolic analysis. The analysis allows us to build a logical formula, which captures multiple program executions, and leverage first-order reasoning to statically prove program properties. The analysis is based on the symbolic semantics, as shown in Fig. 1.16, which is very similar to the concrete semantics. The program is

$$\frac{(e,\Delta) \longrightarrow e^s}{(\texttt{out}(e),\phi,\Delta) \overset{(\phi,e^s)}{\rightarrow} (\texttt{skip},\phi,\Delta)} \qquad \frac{(e,\Delta) \longrightarrow e^s}{(x := e,\phi,\Delta) \rightarrow (\texttt{skip},\phi,\Delta[x \mapsto e^s])}$$

$$\frac{(P_0,\phi,\Delta) \overset{(\alpha)}{\rightarrow} (P_0',\phi',\Delta')}{(P_0; P_1,\phi,\Delta) \overset{(\alpha)}{\rightarrow} (P_0'; P_1,\phi',\Delta')} \qquad \frac{(b,\Delta) \longrightarrow b^s,\ \phi' = (\phi \wedge b^s),\ \phi' \textit{ consistent}}{(\texttt{if } b \texttt{ then } P_0 \texttt{ else } P_1,\phi,\Delta) \rightarrow (P_0,\phi',\Delta)}$$

$$\frac{}{(\texttt{skip}; P_1,\phi,\Delta) \rightarrow (P_1,\phi,\Delta)} \qquad \frac{(b,\Delta) \longrightarrow b^s,\ \phi' = (\phi \wedge b^s),\ \neg\phi' \textit{ consistent}}{(\texttt{if } b \texttt{ then } P_0 \texttt{ else } P_1,\phi,\Delta) \rightarrow (P_1,\neg\phi',\Delta)}$$

Figure 1.16: Symbolic Semantics

executed on symbolic inputs and, consequently, the state is also symbolic. Initially, program identifiers are mapped to fresh variables. We use e^s to range over symbolic expressions, which are build over these initial variables and constants using the standard machinery. A symbolic state is a tuple (P, ϕ, Δ), where P is the program text, ϕ is the path condition and Δ is the symbolic store, mapping identifiers to symbolic expressions. A path condition ϕ is a symbolic boolean expression built over the initial variables and constrains the set of concrete initial states that execute the path. For instance, the symbolic state $(P, (\texttt{l} + \texttt{k} \geq \texttt{0}), \Delta[l \mapsto \texttt{k} + \texttt{1}])$ denotes the program text P, the path condition $(\texttt{l} + \texttt{k} \geq \texttt{0})$ and the symbolic store Δ, which maps the program identifier l to the symbolic expression $\texttt{k} + \texttt{1}$. The rules in Fig. 1.16 can be used to extract a symbolic program model which only contains the information needed to verify the security property. We call such model a *symbolic output tree* (SOT). Broadly, a symbolic algorithm does the following in a loop until all program paths are explored: it starts with symbolic values for input identifiers and executes the program symbolically by applying the rules in Fig. 1.16. When a conditional statement is reached, the consistency of the resulting path condition is checked in order to make sure that the path is reachable. When an output statement is reached, the corresponding output expression is also evaluated in the symbolic state. The symbolic output tree represents conditions on initial inputs that direct the program to an output statement. This is done by saving the path conditions and the output expressions for all reachable output statements. Fig. 1.17 shows the symbolic output tree derived from the symbolic analysis of the program in Example 1.4.1.

The symbolic output tree S and the noninterference formula are combined into a quantifier-free formula, which can be later verified with an SMT (Satisfiability Modulo Theory) solver. Let $N(S)$ denote the set of the tree nodes, and ϕ_n and

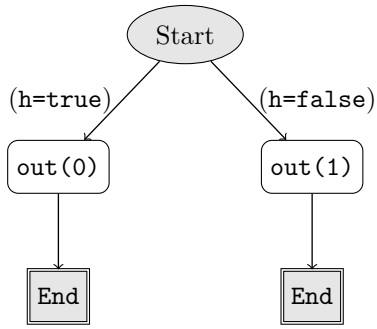


Figure 1.17: Symbolic Output Tree

$\mathtt{O}_n$ the path condition and the output expression at node n, respectively. Then the program is noninterfering iff the following quantifier-free formula is unsatisfiable.

$$\bigvee_{n \in N(S)} (\phi_n \ \wedge \ (\bigwedge_{m \in N(S)} \neg(\phi'_m \ \wedge \ \mathtt{O}_n = \mathtt{O}'_m))) \tag{1.5}$$

The primed version ψ' of a formula ψ is a renaming of all free variables $\mathtt{x}$ in ψ with $\mathtt{x}'$. Intuitively, if the formula in (1.5) is unsatisfiable, then it is impossible to find a pair of (high) initial stores that lead to two different output sequences, and this is exactly the noninterference condition.

Example 1.4.4 *We apply the condition (1.5) to check whether the symbolic output tree in Fig. 1.17 satisfies noninterference, i.e. the formula is unsatisfiable. Let* $\phi_0 := (\mathtt{h} = \mathtt{true})$, $\mathtt{O}_0 := \mathtt{0}$ *and* $\phi_1 := (\mathtt{h} = \mathtt{false})$, $\mathtt{O}_1 := \mathtt{1}$, *then*

$$\bigvee_{n \in \{0,1\}} (\phi_n \ \wedge \ (\bigwedge_{m \in \{0,1\}} \neg(\phi'_m \ \wedge \ \mathtt{O}_n = \mathtt{O}'_m)))$$

For $n = 0$, *one of the disjuncts has the following shape,*

$$(\mathtt{h} = \mathtt{true}) \ \wedge \ (\neg((\mathtt{h}' = \mathtt{true}) \ \wedge \ \mathtt{0} = \mathtt{0})) \wedge (\neg((\mathtt{h}' = \mathtt{false}) \ \wedge \ \mathtt{0} = \mathtt{1}))$$

The above formula is satisfiable for $\mathtt{h} \mapsto \mathtt{true}$ *and* $\mathtt{h}' \mapsto \mathtt{false}$. *Hence the disjunction is satisfiable and the program is interfering as expected.*

The program in Example 1.4.1 would have been rejected by a security type system as well, for instance using the rules in Fig. 1.13. To further appreciate the advantages of our approach, the reader is encouraged to go back to the beginning of this subsection and apply the same analysis to the following secure program.

$$P' ::= \left[\begin{array}{l} \mathtt{if}\ h\ \mathtt{then}\ \mathtt{out}(0)\ \mathtt{else}\ \mathtt{skip}; \\ \mathtt{if}\ \neg h\ \mathtt{then}\ \mathtt{out}(0)\ \mathtt{else}\ \mathtt{skip} \end{array} \right.$$

1.4.2 Thesis Overview

In the first part of the thesis we show how the epistemic logics can be used to express information flow concepts in language-based and event-based systems and conclude with an algorithmic approach to check program robustness in presence of active attacks. In the second part of the thesis we address the verification problem of information flow security policies including symbolic algorithms and practical tools. The approach is based on model checking and automatic theorem proving techniques to verify a piece of code with respect to a given security policy. We now give a short overview of each paper included in the thesis followed by a statement of the author's contribution.

Epistemic Temporal Logic for Information Flow Security

A common feature in much recent work on information flow analysis has been the appeal to the concept of knowledge as a fundamental mechanism to bring out what security property is being enforced (the "revealed" knowledge) and compare it with the knowledge allowed by the policy. This appeal to knowledge, typically as equivalence relations on initial states (or partial equivalence relations [201]), has been important to produce clear, reference conditions on which soundness arguments can be based. Knowledge, as it happens, is at the root of an entire branch of logic, namely the logic of knowledge, or epistemic logic [112]. In this paper we show that the epistemic logic account of knowledge is compatible with the knowledge notion which has emerged within language-based security, and can have a valuable role to play for policy specification.

Temporal epistemic logic is a well-established framework which can be used in distributed systems to reason about knowledge and how it evolves over time. Temporal epistemic logic adds epistemic connectives K and L to familiar temporal connectives such as G (always) and U (until). Those epistemic connectives, as formalized in the previous subsection, relate agents local state to the possible global states that are consistent with the agents local observations. Thus, as an example, the property $\phi = G(C \rightarrow \forall v.\, L(h{=}v))$ expresses that whenever some condition C holds then, as far as the attacker can tell, any value of h is possible (and so the value of h is unknown and not released to the attacker).

In this study we apply temporal epistemic logic to standard sequential imperative programs [223] augmented with a public output statement, in order to allow a program to "gradually release" [28] information. The program model is turned into a model for temporal epistemic logic in the style of interpreted systems [112]. This is done by defining a perfect recall epistemic accessibility relation using the simple and intuitive idea that two execution states should be regarded as being epistemically the same if they have been reached by identical traces of publicly observable output, i.e. such that an observer cannot tell the two states apart. In particular, if there exists an execution sequence producing a trace τ and ending in a state refuting property ϕ then the attacker is forced to hold $\neg\phi$ for possible.

Our main objective with this paper is to show that temporal epistemic logic is an interesting and natural device with which to express information flow policies. We show this by demonstrating how various state-based security conditions related to noninterference [122, 123] (absence of "bad" information flows) and declassification [202] (intended release of information) can be characterized using the logic.

Statement of Contribution. This paper is published in the *Proceedings of the ACM SIGPLAN 6th Workshop on Programming Languages and Analysis for Security* (PLAS 2011). The paper is coauthored with Mads Dam and Gurvan Le Guernic. Initially, Mads proposed to use epistemic logic for information flow properties. Then we all contributed equally to the technical development and the paper writing. I wrote most of the formal proofs.

A Logic for Information Flow Analysis of Distributed Programs

Information flow security policies, if successfully enforced or verified, prevent different types of confidentiality and integrity attacks. However, most work on language-based security models of information flow assumes synchronous or relational communication [100, 197, 182]. Although these models are important in many settings, they are not obviously well suited for distributed programs where communication is interactive/reactive, nondeterministic and mostly asynchronous, lossy or unordered. The result is that programs that are considered insecure in one model may be secure in another, and vice versa. The following scenario illustrates the need for security conditions that go beyond the traditional initial-state final-state account of noninterference.

An online auction is a distributed system consisting of an auctioneer A and several bidders B_i competing for items I_k. Such systems are complex and usually involve both message passing and shared memory. For example, the auctioneer may receive messages from bidders who want to participate in the auction and associate a dedicated thread to each request. Then, depending on the auction protocol, each thread may read and write to a private shared array containing bids for all bidders and items. Several information flow policies may be worth enforcing in this scenario.

P_1 : The authentication code (pwd) of bidder B_i is always (G) secret wrt. any bidder B_j. In logic: $G\neg K_{B_j}(pwd_{B_i} = v)$.

P_2 : The sequence of bids of bidder B_i remains secret wrt. all bidders B_j until (W) the auction is closed. In logic: $L_{B_j}(secArray = v)\ W\ aClosed$.

P_3 : Only the first 3 bids of bidder B_i are considered secret wrt. any bidder B_j until the auction is closed. In logic: $L_{B_j}(\phi(b_1^i, b_2^i, b_3^i))\ W\ aClosed$.

P_4 : Any bid of bidder B_3 remains secret wrt. a colluding attack of B_1 and B_2. In logic: $GL_{B_1,B_2}(b^3 = v)$.

P_5 : The system may nondeterministically select a subset of bids from the private array, compute the maximum and promote an item I^* to the winner B^*. The output of this process may be considered secret wrt. any bidder $B_j \neq B^*$. In logic: $G\neg K_{B_j}(out(B^*, I^*))$.

As illustrated above, the auction system poses several challenges that need to be addressed to enforce the security policies. First, the system is inherently nondeterministic, hence the need for possibilistic notions of information flow security. Second, distributed programs are usually interactive/reactive, which requires protection of sequences of (input or output) events as opposed to classical relational models where the input is read in the beginning of execution. Third, security policies are usually dynamic and involve controlled release of secret information. Finally, in distributed settings attackers may collude and share their observations to disclose secret information.

In this paper we model systems in a trace-based setting where an execution trace is a sequence of events on channels. Security properties are expressed in terms of epistemic conditions over system traces. Noteworthy, the security model brings out what events $\mathcal{O}$ on channels an observer can see and what observations on events $\mathcal{P}$ should be protected. Then the system is secure if the knowledge about events in $\mathcal{P}$ of an observer who makes observations in $\mathcal{O}$, at any point in the execution trace, is in accordance with the security policy at that point. Namely, the observer is unable to learn more information than what is allowed at a given point while moving to a successive point of the same trace and possibly making a new observation. This model fits well with current knowledge-based approaches to information flow security and, by being explicit about the information that needs to be protected, it allows a very general treatment of secret information, both as high level input and output events, and as relationships between events, say ordering, multiplicity, and interleaving. We show that several possibilistic conditions such as separability, generalized noninterference, nondeducibility, nondeducibility on outputs and nondeducibility on strategies are accurately reflected in the epistemic setting. Moreover, we present a linear time epistemic logic, with past time operators, which allows us to syntactically characterize security properties. The logic can be used as specification language for expressing possibilistic information flow policies. This enables modeling of the intricate and precise policies described in the example and, at the same time, ensures separation between the actual code and the policy.

Statement of Contribution. This paper is an extended version of the work published in the *Proceedings of the 18th Nordic Conference on Secure IT Systems* (NordSec 2013). This is my own paper, however Mads had an advisory role in all stages of this work.

A Weakest Precondition Approach to Robustness

In this paper we study the interaction between integrity and confidentiality, and show how low integrity, i.e. attacker-controlled, code can compromise the program confidentiality and enable the attacker to learn more information than it is allowed to. According to OWASP (Open Web Application Security Project) [10], the most common security attacks are due code injections, which modify the intended program semantics and unduly send sensitive data to the attacker. In language-based security, this problem is known as robust declassification [227] and it usually arises whenever the program code explicitly downgrades sensitive information or upgrades untrusted information. For confidentiality, the goal is prevent untrusted code from influencing the decision to declassify secret data and thus reveal more information than intended. More generally, a program is robust if an active attacker, who sends untrusted code to the program, is unable to learn more sensitive information than a passive attacker who can only observe the public outputs of the program. For example, the Javascript code snippet in Fig. 1.18 can be used by the attacker to send the user's cookie to a web server under her control.

```
/* initialisation of the cookie by the server */
   var cookie = document.cookie;
   var dut;
   if (dut == undefined) {dut = "";}
   while(i<cookie.length) {
       switch(cookie[i]) {
         case 'a': dut += 'a'; break;
         case 'b': dut += 'b'; break;
         ... }
   }
/** dut now contains a copy of cookie;
    when the user clicks on the image, dut is sent
    to the web server under the attacker's control
*/
document.images[0].src = "http://badsite/cookie?" + dut;
```

Figure 1.18: An XSS vulnerability.

We approach this problem by analyzing attackers that can observe the input/output (I/O) behavior of programs and that, from these observations, can make some kind of *reverse engineering* in order to derive the secret information that could have possibly produced the observations. This idea of *backward analysis* for noninterference and declassification is studied in the context of abstract interpretation framework [42]. The ingredients of this method are: the initial declassification

policy modeled as an abstraction of private input domain and the weakest liberal precondition semantics of the program, characterizing the backward analysis (i.e., from outputs to inputs) and the attacker's observational power. The certification process starts with a possible public observation and computes the weakest liberal precondition of the program and the observation. By definition, the *weakest* precondition semantics provides the *greatest* set of possible input states leading to the given output observation. In other words, it characterizes the greatest collection of input states, and in particular of private inputs, that an attacker can identify starting from the given observation. In this way, the attacker can restrict the range of private inputs inside this collection and possibly reveal secret information. The analysis of the code in Fig. 1.18 will produce the following result (assuming that the variable *cookie* is labeled as high security and the variable *dut* is labeled as low security).

$$
\begin{array}{l}
[\bullet] \\
\{cookie + dut = res\} \\
while(i < cookie.length)\{ \\
\quad switch(cookie[i])\{ \\
\quad case'a' : dut+ =' a'; break; \\
\quad case'b' : dut+ =' b'; break; \\
\quad \ldots\}\} \\
\{dut = res\}
\end{array}
$$

The resulting formula shows that confidentiality of *cookie* is violated since there is an implicit flow towards the public variable *dut*. Consequently, this is the sensitive information disclosed by a passive attacker whenever *dut* is initialized with the empty string.

Nevertheless, if the variable *dut* is labeled as low integrity, an active attacker can insert malicious code and thus leak additional confidential information. For instance, if the attacker is interested in the *history* object, she can loop over the elements of the object and reveal through variable *dut* all the web pages the client has had access to. The code in Fig. 1.19 can be a possible malicious injection at [•] point in the example.

```
<script language="JavaScript">
var dut = "";
for (i=0; i<history.length; i++){
    dut = dut + history.previous;
}
</script>
```

Figure 1.19: Malicious code exploiting XSS vulnerability.

As a result, the program clearly violates the robustness condition. In this work,

our objective is to formalize the analysis also in presence of active attackers. We consider the model of active attackers which can transform program semantics by inserting untrusted code in fixed program points [•] (*holes*), known by the programmer. We then show that the weakest precondition calculus can be used to characterize the revealed information, and therefore to discover program vulnerabilities. The analysis is thus used to certify program robustness. Since the possible active attacks can be infinitely many, we explore them symbolically and provide *sufficient conditions* that guarantee robustness independently of the attack. Initially we study robustness for I/O attackers, i.e., attackers that can only observe the I/O program behavior, and afterwards we extend the method to attackers able to observe intermediate states, i.e., the trace semantics of the program. In some restricted contexts, for example where the activity of the attacker is limited by the environment, the standard notion of robustness may become too strong. To deal with these situations we introduce a weakening of robustness, i.e., *relative robustness*, which restricts the set of active attackers that we are checking robustness for. We conclude with various interesting applications where the approach can be useful in order to capture confidential information leakages.

Statement of Contribution. This is a journal paper published in the *Transactions on Computational Science X: Special Issue on Security in Computing, Part I* (TCS 2010). The paper is coauthored with Isabella Mastroeni. I was responsible for most of the technical results, proofs and paper writing, while Isabella had an important advisory role. An earlier version of this paper is published in the *Proceedings of the ACM SIGPLAN 4th Workshop on Programming Languages and Analysis for Security* (PLAS 2009).

ENCoVer: Symbolic Exploration for Information Flow Security

Epistemic logic, the logic of knowledge, provides a clean and intuitive tool for modeling different information flow policies, including noninterference and many variants of declassification, as in a number of recent works [38, 128, 28, 45]. The knowledge of an attacker that is in possession of the program text and has a partial view of program executions, e.g. by receiving some outputs, can be defined as a partition of the set of secret inputs that determines the observed outputs. This partition corresponds to the properties of secret inputs disclosed by the program. The desired security policy, e.g. some noninterference or declassification property, gives rise to another partition of secret inputs, the property of secret inputs allowed to flow to the observer. Comparing these two partitions determines whether the program meets the security policy. In epistemic logic, the observer's knowledge is expressed in terms of knowledge operator $K\phi$, meaning that the observer knows property ϕ i.e. ϕ is true in all states that are possible given the observer's current state [38, 112]. Intuitively, $K\phi$ holds for all formulas ϕ that induce a partition which is less discriminating (included into) than the one induced by the observed outputs.

Many verification techniques have been proposed for checking information flow properties, including static and dynamic analysis [197]. Security type systems [218, 139] is the dominant technique, but other techniques have been explored as well, including dependency analysis [14], program logics [45], abstract interpretations [119], axiomatic approaches [21], program slicing [220] and so on. Most verification approaches for noninterference-like policies, type systems in particular, enforce noninterference by separating the secret and public computations, and as a consequence any interaction between the secret and public computations, even a benign or corrective one, deems the program as insecure. This increases the number of false positives and limits applicability. Other techniques are based on semantical reasoning and are often computationally expensive or even undecidable. The verification approach proposed in this paper is exclusively tailored to end-to-end verification of noninterference and declassification by means of off-the-shelf epistemic model checkers and SMT solvers. Thereby, the approach is both sound and relatively complete with respect to verification in the underlying program model.

In this paper, concolic execution, a mix of concrete and symbolic execution, is used to extract a bounded model of program runtime behavior [121, 146]. This model is subsequently verified against the target security properties, expressed in epistemic logic, by means of an epistemic model checker. Due to the size of the input data domain epistemic model checking can, however, be extremely inefficient or even infeasible. To address this, an alternative approach is proposed whereby the model checking problem is transformed to a first order logic formula. Due to the shape of epistemic formulas for noninterference and declassification, the transformation produces a formula which only contains existential quantifiers, thereby an SMT solver can be used to perform the checking efficiently.

We have implemented the verification approach described above in a tool prototype, ENCoVer [39], as reported in Fig. 1.20. The prototype is an extension of Java PathFinder [184], a software model checker developed at NASA. ENCoVer takes as input a program written in Java and a security policy and generates a *symbolic output tree*, which encodes conditions on program inputs that produce output observations. The symbolic output tree is used in two ways. First, it is combined with the security policy to generate an SMT formula which is subsequently verified with Z3, a state-of-art SMT solver [98] and, secondly, as an alternative, it is used to generate an input file for the epistemic model checker MCMAS [157]. The performance of ENCoVer is evaluated on a main case study involving multiple parties accessing a joint store of tax records, as well as on several smaller, but delicate, examples.

Statement of Contribution. This paper is published in the *Proceedings of the 25th IEEE Computer Security Foundations Symposium* (CSF 2012). The paper is coauthored with Mads Dam and Gurvan Le Guernic. All authors contributed equally to the technical development and the paper writing. Gurvan was the driving force behind ENCoVer, although we both contributed to the tool implementation

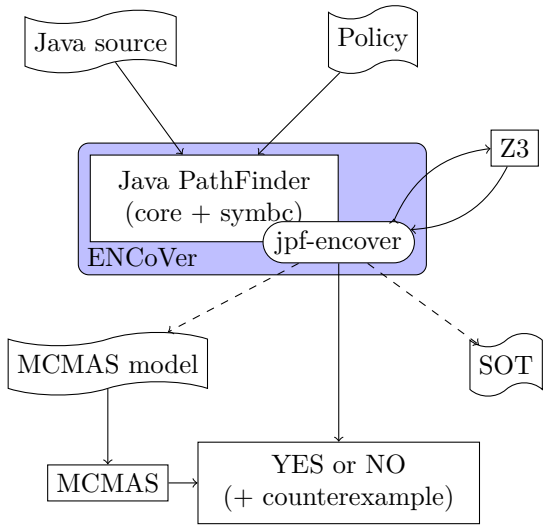


Figure 1.20: ENCoVer Architecture

and the case studies. The formal proofs in the paper were written by myself.

Automating Information Flow Analysis of Low Level Code

The ultimate goal of information flow analysis is to establish confidentiality and integrity properties of real code executing on commodity CPUs. In the literature, normally this problem is addressed at the source code level. There it may be more forgiving to ignore messy low-level problems, e.g. regarding timing, complex control flow, or hardware specifics. Also, one may appeal to special compilers that avoid difficult optimizations, or work around machine features such as caching, instruction reordering, concurrency, I/O, interrupts, bus contention and so on, that are difficult to handle in a precise manner.

Sometimes, however, source level analysis is less suitable. This is certainly the case when dealing with third-party code, but it applies in other cases too, for instance, for heavily optimized or obfuscated code, and for kernel handler routines that manipulate security sensitive peripherals such as privileged processor registers, MMUs, and bus and interrupt controllers.

The literature has two "standard" approaches to information flow control (IFC) for low-level languages: static and dynamic verification techniques [197]. Neither of these schools are very helpful, though, when it comes to the problem we have set out to study: Information flow analysis for low level code on commodity processors. In this domain, existing static approaches are too imprecise due to lightweight (data/flow/path/timing-insensitive) analysis, while dynamic approaches suffer from the well known problem of label creep and introduce undesired runtime overhead.

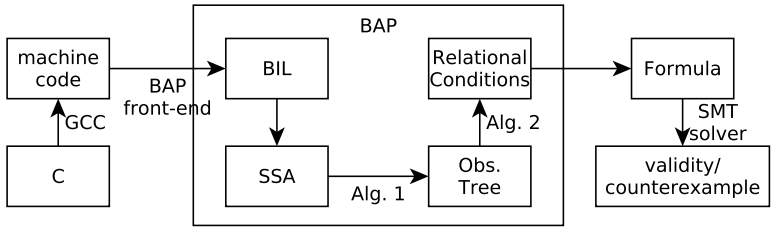


Figure 1.21: Verification process

Security testing-like techniques provide impressive results in terms of scalability, however, they are in general unsound and can not directly be used for full verification [180, 31].

In this paper we propose to directly verify relational information flow properties at machine code level, leveraging as much as possible recent progress on low-level code analysis tools such as BAP [62]. Code for our target machine, ARMv7 assembly, is first lifted to a machine-independent intermediary form, BIL, using the BAP tool. This process uses a lifter that is produced from the Cambridge HOL4 model of ARMv7 [135]. This allows the reuse and extension of BAPs program verification back end to symbolically execute the resulting BIL code.

Fig. 1.21 depicts the workflow of the verification process. We first perform *unary* analysis (Alg. 1) and then verify relational properties by propagating relational preconditions through each of a pair of related programs until a pair of observation points are reached, that need to be matched, in order for the relational property to hold (Alg. 2). These observation points are memory write events, to locations that are statically determined to be observable by some external observer, because of multithreading, or memory-mapped I/O, or for some other reason. Matching is done by SMT solving using STP [118], on formulas that tend to grow huge, but generally rely only on linear arithmetic, uninterpreted functions, and arrays, and so are not too costly to check. Special care is needed for memory accesses which introduce quantifier alternation, hence we propose an instantiation technique which ensures the resulting formulas are quantifier free.

Three distinguishing features make our information flow analysis both useful and challenging: *loop invariants*, *timing* and *traces*. Loops are handled using (relational) invariants/widening. We point out that relational invariants are, probably counter-intuitively, simpler than state invariants as in many cases they do not require proving functional correctness of the loop. Our case studies show that the invariants we provide are conjunctions of linear equalities, which can be generated automatically [206]. Timing is particularly critical. The timing information is included in the symbolic state and propagated with the other constraints. The model used here scales to functional cost models, i.e. models where the timing cost can be calculated as function of the input instruction, independent of the history. This is evidently realistic only for simple processor architectures such as ARM Cortex-M

(but we note that a vast number of such processors are in use today in critical control applications). Richer and tractable timing models that can take into account also features like caches and instruction pipelines are, however, currently not available at ISA level. Finally, the trace-based analysis broadens the number of target applications handled by our technique, including preemptive environments and scheduling.

In general the approach will suffer from scalability problems, for instance due to path explosion, and due to the generally complex and detailed machine state. However, our primary application is separation kernel handler verification, and this domain is generally characterized by critical machine code fragments that are rather small (generally under 1K instructions per handler), but also tricky. The case studies reported in this paper are based on syscall handlers and device drivers of slightly more than 250 lines of ARMv7 assembly, produced by a mix of hand-crafted assembly and GCC-optimized C.

Statement of Contribution. This paper is accepted and will appear in the *Proceedings of the 21st ACM Conference on Computer and Communication Security* (CCS 2014). The paper is coauthored with Mads Dam and Roberto Guanciale. Mads and I initially developed the idea of automating relational verification for low level code. We all contributed equally to the technical development and the paper writing. The tool prototype was developed by myself.

1.5 Concluding Remarks

This thesis contributes to the state of the art both in terms of theoretical foundations and practical developments. The use of the logic of knowledge captures in a very clean way the epistemic underpinnings inherent in the information flow security conditions, as described in Chapter 2 and Chapter 3. Moreover, the delicate intermingle of confidentiality and integrity, as addressed in Chapter 4, results fundamental in securing modern applications and our techniques move a step forward in that direction. The verification methods proposed in the last two chapters are tailored to information flow properties and, as a result, they provide a precise enforcement of the security conditions. Namely, whenever the verification process terminates, it will either certify the program as secure or report a violation which is a real bug. On the practical side, the algorithms and the tools presented in Chapter 5 and Chapter 6 show that information flow analysis is not only necessary and useful, but also possible for real applications, ranging from Java programs to low level ARMv7 assembly.

There are a few issues that need to be addressed before the techniques studied in this thesis get adopted in software production systems. As usual, the migration to new technologies comprises both technical and non-technical challenges which very often limit the use of these technologies. First and foremost, current programming practices mostly consider security as a posteriori task and rarely embed the security

requirements in all phases of the software development process. Changing this trend is quite challenging as a security-aware software development process would require a mindset shift for all the stakeholders involved in the process, including software architects, programmers, analysts and even the end-users. This leads to additional costs and efforts which can certainly pay off in the long run, however not everyone is willing to take on these expenses. On the other hand, software bugs are at the root of many security failures and ad-hoc security patches only increase the overall complexity and can make the software more error-prone. Therefore, the adoption of systematic methods for security analysis and certification can provide strong formal guarantees and thus significantly reduce the space of possible attacks. The achievement of this goal may require to bridge the usual gap between research and industry, which we hope it will happen in the future.

Moreover, the appealing end-to-end security assurance provided by IFC comes at the price of security policies which, at least compared to traditional access control, are not very simple and may require reasoning about the whole system. This leads to a computationally expensive verification process which sometimes may not give a satisfactory answer or it may constrain the way a program is written at best. From the theory perspective we can arguably say that IFC stands on a solid ground, although some challenges, as discussed in Sect. 1.3.2, still remain. More needs to be done in terms of practical tools that help the developers to program with information flow control. Below we discuss a few open issues regarding the work presented in this thesis and propose several directions that can be taken to tackle these problems in the future.

Tools and Case Studies. The tools developed in this thesis are research prototypes used to support the theoretical results. More engineering work would certainly improve the efficiency of the tools and allow them to scale to larger case studies. The verification algorithms rely heavily on symbolic execution and SMT solving and, as a result, known limitations of these methods apply to our tools as well. For instance, ENCoVer, being an extension of JPF, is currently limited to the class of programs that SPF (the symbolic extension of JPF) can handle and the class of expressions the SMT solver (Z3) can solve. In principle SPF can execute any Java bytecode, however in practice SPF is limited by missing implementations for some native libraries (such as java.io and java.net), a few bugs (such as NullPointer exceptions being reported as NoSuchMethod exceptions), and the state space explosion problem (specially for multithreaded programs with loose synchronization constraints). Similarly, the ARMv7 information flow analysis tool extends the CMU Binary Analysis Platform (BAP) to implement the symbolic algorithms and uses the STP solver to verify the resulting constraints. In the future, we expect the class of programs handled by our tools to grow due to the active development of JPF and BAP as well as the advances in the state of the art SMT solvers. We also believe that the tool development process should be accompanied by realistic case studies and benchmarks which bring out what types of program analysis and verifications

are needed. The case studies considered in this thesis move a step in this direction, however much more can and should be done.

Abstraction. The algorithms presented in the thesis are tailored to precise verification of information flow properties. As a result, the verification process turns out to be computationally expensive and sometimes even infeasible. For instance, program loops, which appear in many real applications, are either ignored (by bounding the number of loop iterations cf. ENCoVer) or loop invariants are provided manually by the programmer (cf. the ARMv7 tool). The case studies show that (relational) loop invariants tend to be simple, which makes it possible to generate, and likely also verify, automatically in the future.

Moreover, some applications may not always need a precise information flow analysis, at least not for the whole program. Sound approximate analysis, for instance type systems or predicate abstraction, have been successfully applied to combat the state explosion problem. An interesting line of future work is to combine these lightweight analysis, e.g. a security type system, with our techniques in order to get the best of both approaches, namely increase scalability while maintaining precision.

Refinement. It is well-known that information flow properties are properties of sets of executions, i.e. hyperproperties, and in general such properties are not preserved under refinement. Hence, adding and removing executions as a result of approximations can make the analysis unsound or imprecise. The condition of observational determinism used in Chapter 6 is known to be preserved under refinement, however, in presence of nondeterminism, it turns out to rule out useful programs. An interesting direction of future work is to relax observational determinism and study weaker conditions that preserve the information flow properties under refinements. This is important as it would enable techniques for stepwise development of software with information flow security guarantees.

Usability. The use of epistemic logics for policy specification requires the programmer to know about these logics in order to write the security policies. This may intimidate programmers that are not experts in logics. Much work can be done to improve the usability of the tools. For instance, we can use high level formalisms, e.g. diagrams, to intuitively describe the meaning of the logic connectives and provide templates for security policy specifications using these folmalisms. In addition, the integration with existing development environments (IDE:s), comprehensive error reports and documentation can certainly help the programmers and increase usability.

Composition and Decomposition. The knowledge-based security conditions studied in Chapter 2 and Chapter 3 are stated as global conditions over system

models. While this characterization may be needed in the worst case, some applications allow to decompose the system model and the security policy, and thus make the verification of the epistemic properties easier. In particular, one can leverage properties of the target systems, say asynchrony or message order, and decompose the analysis over simpler systems and properties.

Compositionality is also a desirable feature to increase scalability. We can annotate the program with *local* knowledge-based specifications and later combine these specifications to entail some global knowledge-based specification. For information flow security, this approach would enable a compositional security analysis of the target application by combining components which are shown to be secure locally.

Robustness. In Chapter 4 we present an algorithmic approach to program robustness using the weakest precondition calculus. The analysis is fully static and it requires a well-defined language semantics to be applied. The static nature of this approach paddles against the dynamic nature of our target languages, for instance Javascript. Hence, a combination of static and dynamic security checks may be needed during the implementation phase of this work. Moreover, effective algorithms and tool support are also left out as future work.

Part I

Specification

Chapter 2

Epistemic Temporal Logic for Information Flow Security

Musard Balliu and Mads Dam and Gurvan Le Guernic

Abstract

Temporal epistemic logic is a well-established framework for expressing agents knowledge and how it evolves over time. Within language-based security these are central issues, for instance in the context of declassification. We propose to bring these two areas together. The paper presents a computational model and an epistemic temporal logic used to reason about knowledge acquired by observing program outputs. This approach is shown to elegantly capture standard notions of noninterference and declassification in the literature as well as information flow properties where sensitive and public data intermingle in delicate ways.

2.1 Introduction

Information flow analysis and language-based security has been a hot topic for well over ten years now. A large array of specification and validation techniques have been proposed, involving security properties (multi-level security, mandatory access control), semantical modeling techniques (trace conditions, simulations and bisimulations/unwinding conditions), and analysis and enforcement techniques (type systems, dependency analyses of various forms). A critique that may be leveled at much of the past work, our own included, is that it has not always managed to separate concerns very clearly. In particular, constraints in specification techniques, programming language features, and details and limitations in the enforcement/-analysis mechanisms have been interdependent in such a way that it has often been

unclear exactly what properties are enforced and how the various approaches relate to each other. Also, as pointed out by several authors [44, 191], the policy specification mechanisms have often been interwoven with the object (the program) on which the policy is to be enforced in a manner that makes it hard to separate policy concerns from enforcement concerns.

A common feature in much recent work on information flow analysis, cf. [28, 44, 191], has been the appeal to the concept of *knowledge* as a fundamental mechanism to bring out what security/confidentiality property is being enforced (the "revealed" knowledge) and compare it with the knowledge allowed by the policy. This appeal to knowledge, typically as equivalence relations on initial states (or partial equivalence relations [201]), has been important to produce clear, external reference conditions on which e.g. soundness arguments can be based. Knowledge, as it happens, is at the root of an entire branch of logic, namely the logic of knowledge, or epistemic logic. In this paper we aim to show that the epistemic logic account of knowledge is compatible with the knowledge notion which has emerged within language-based security, and can have a valuable role to play for policy specification.

Temporal epistemic logic is a well-established framework [112] which can be used in distributed systems to reason about knowledge and how it evolves over time. Temporal epistemic logic adds epistemic connectives K and L to familiar temporal connectives such as G (always) and U (until). Those epistemic connectives relate agents local state to the possible global states that are consistent with the agents local observations. The property $K\phi$ expresses that an agent A observing a program "knows" ϕ in the sense that ϕ holds in all states that are possible given A's past observations. Dually, $L\phi$ expresses that *some* observationally equivalent state exists for which ϕ holds. Thus, as an example, the property $\phi = G(C \rightarrow \forall v.\, L(h{=}v))$ expresses that whenever some condition C holds then, as far as the attacker can tell, any value of h is possible (and so the value of h is unknown and not released to the attacker).

In this study we apply temporal epistemic logic to standard sequential while programs augmented with a public output statement, in order to allow a program to "gradually release" [28] information concerning its initial state. The program model is turned into a model for temporal epistemic logic in the style of interpreted systems [112]. This is done by defining an S5 perfect recall epistemic accessibility relation using the simple and intuitive idea that two execution states should be regarded as being epistemically the same if they have been reached by identical traces of publicly observable output, i.e. such that an observer cannot tell the two states apart. In particular, if there exists an execution sequence producing a trace τ and ending in a state refuting property ϕ then the attacker is forced to hold $\neg\phi$ for possible.

Our main objective with this paper is to show that temporal epistemic logic is an interesting and natural device with which to express information flow policies for imperative programs. We show this partly by example, and partly by demonstrating how various state-based security conditions related to noninterference [122, 123] (absence of "bad" information flows) and declassification [202] (intended release of

information) can be characterized using the logic.

We are not the first to apply epistemic logic in the context of computer security. The concrete link between language-based security and temporal epistemic logic which we point out in this paper appears, however, to be new. BAN logic [68] and successors use epistemic concepts to model agents changing knowledge and belief in security protocols. BAN logic, however, suffered from a lack of an intuitively acceptable semantics (the problem of logical omniscience), something that has only been remedied recently [88]. Post-BAN work in security protocol verification has to a large extent focused on Dolev-Yao types of direct knowledge extraction. This approach works well for many concrete protocols, but it is not adequate to capture the types of indirect channels of high importance in language-based security. For formal analysis of distributed protocols and multi-agent systems, epistemic logic and various extensions have extensive histories [112]. Much recent work in the area has focused on model checking [187, 116]. Applications of epistemic concepts have been made in process calculi such as the applied π-calculus [73] and CCS [161] and to model protocols for instance in the area of electronic voting [54]. A precursor of our approach is Askarov and Sabelfeld's gradual release model [28] where attackers knowledge is modeled as equivalence relations on initial states. In the paper we look into this relationship in more detail and show how gradual release and a number of other possibilistic state-based security conditions can be characterized using temporal epistemic logic.

In Section 2.2 we set up the underlying computational model. Section 2.3 introduces the syntax and semantics of linear time temporal epistemic logic on these models, and shows how the model relates to the standard interpreted systems model [112]. We then turn to various well known security conditions from the literature, including noninterference and different flavors of declassification along the dimensions considered by [202] in Sect. 2.4 to 2.7. We finally point out some open issues and directions for future work.

2.2 Computational Model

In this section we set up our language's basic computational model. We study a simple while language extended with a synchronous output statement that, over the course of a computation, causes information to be leaked to an observer. Besides the output statement "$\texttt{out}(e)$", the features of our while language are commonplace: assignments, conditionals, while loops, a primitive data type of values belonging to a finite set *Val*. The grammar of the language is given in Fig.2.1. Programs are ranged over by P, identifiers by x, values by v, and expressions by e.

A store is a finite map $\sigma : x \mapsto v$, and $\sigma(e)$ is the value of e in store σ. An execution state is a pair (P, σ). The execution of a program generates observable actions (or events) belonging to Act and ranged over by α ($Act = \{\texttt{out}(v) \mid v \in Val\}$). The transition relation $(P, \sigma) \xrightarrow{\alpha} (P', \sigma')$, or $(P, \sigma) \rightarrow (P', \sigma')$, states that by taking one execution step in the execution state (P, σ) the execution generates

$$P ::= \texttt{skip} \mid \texttt{out}(e) \mid x{:=}e \mid P_1 \texttt{ ; } P_2 \\ \mid \texttt{if } e \texttt{ then } P \texttt{ else } P \mid \texttt{while } e \texttt{ do } P$$

Figure 2.1: Programming language grammar

the visible event α, if it is present, and the new execution state is (P', σ'). We write $(P, \sigma) \xrightarrow{(\alpha)} (P', \sigma')$ where α is optional.

Definition 2.2.1 (Execution)
An execution *is a finite or infinite sequence of execution states.*

$$\pi = (P_0, \sigma_0) \xrightarrow{(\alpha_0)} \cdots \xrightarrow{(\alpha_{n-1})} (P_n, \sigma_n) \xrightarrow{(\alpha_n)} \cdots \tag{2.1}$$

The execution π is maximal *if π is a prefix of the execution π' only if $\pi = \pi'$.*

We write $len(\pi)$ for the length (number of transitions) of the execution π. An *execution point*, or simply *point*, is a pair (π, i) where $0 \leq i \leq len(\pi)$. An execution point (π, i) represents the state of the execution π after i steps. We write $trunc(\pi, i)$ for the prefix of π up to, and including, execution state (P_i, σ_i), the i^{th} execution state of π. We extend the notations as follows: $\pi(i) = (P_i, \sigma_i)$, $P(\pi, i) = P_i$ and $\sigma(\pi, i) = \sigma_i$.

In our model, the power of the attacker is modeled by providing a function *trace* mapping execution points to traces that represent what the attacker has been able to observe so far. In particular, $trace(\pi, i)$ can span from the truncation function $trunc(\pi, i)$ for the strongest attacker able to see all the internal computation, to the function returning the last event generated for a weak memory-less attacker. For the standard noninterference attacker able to observe a set of identifiers X during the execution, *trace* is the function returning the sequence of stores σ_j $(0 \leq j \leq i)$ restricted to the domain X and where identical consecutive stores are collapsed. In the remaining of this paper, we use the function *trace* given in Def. 2.2.2. This definition corresponds to the perfect recall attacker, i.e. only able to observe outputs and having memory of past observations.

Definition 2.2.2 (Trace)
A trace *τ is an element of Act^*. $trace(\pi, i)$ is the sequence of events α_j such that $0 \leq j < i$ and α_j exists. The definition of trace is trivially extended to executions, such that $trace(\pi) = trace(\pi, len(\pi))$*

The trace of the execution (2.1) is: $(\alpha_0)(\alpha_1)\cdots(\alpha_n)\cdots$

A model $\mathcal{M}$ is a set of maximal executions. Normally we take as a model the set of all maximal executions originating from some designated set of initial states, for instance of the shape (P_0, σ_0) where P_0 is a fixed initial program. We write

$\mathcal{M}(P)$ for the set of all maximal executions started at all initial states (P, σ_0) for all initial value stores σ_0. An *epoch* is a set of points reachable by observing a given trace, i.e. $\mathcal{M}$ is implicit,

$$epoch(\tau, \mathcal{M}) = \{(\pi, i) \mid \pi \in \mathcal{M}, 0 \leq i \leq len(\pi), trace(\pi, i) = \tau\}$$

The epoch of a trace τ precisely captures the knowledge obtained by observing τ (in the present possibilistic setting, and ignoring lower level features induced by compilers and run-time systems). For instance, if all points $(\pi, i) \in epoch(\tau, \mathcal{M})$ have the property that the store at that point assigns to x a value between 3 and 5, say, then this fact is known to the observer once she has observed the trace τ. In other words, epoch induce a relations of "equivalent knowledge". Indeed, epochs induce on points a standard epistemic S5 modal accessibility relation $\sim$ by the condition:

$$\begin{aligned} & (\pi, i) \sim (\pi', i') \\ \Leftrightarrow\ & (\pi, i) \in epoch(\tau, \mathcal{M}) \text{ implies } (\pi', i') \in epoch(\tau, \mathcal{M}) \\ \Leftrightarrow\ & trace(\pi, i) = trace(\pi', i') \end{aligned}$$

2.3 Linear Time Epistemic Logic

Reflecting the temporal and epistemic structure of models, we propose to use temporal epistemic logic to express dynamic information flow properties of programs. Many such logics have been considered in the literature [112]. Here we propose to work with a standard, very general and expressive logic in the family of temporal epistemic logics, namely the linear time temporal epistemic logic KL_1 without the *Next* operator, in this paper referred to as $\mathcal{L}_{KU}$.

Definition 2.3.1 (Syntax of $\mathcal{L}_{KU}$)
The language $\mathcal{L}_{KU}$ of formulas ϕ, ψ in linear time temporal epistemic logic is given as follows:

$$\phi, \psi ::= e_1 = e_2 \mid init_x(e) \mid \phi \wedge \psi \mid \neg\phi \mid K\phi \mid \phi U \psi$$

Besides boolean identities $(e_1 = e_2)$, the language contains additional atomic propositions $init_x(e)$ expressing that the value x in the initial state is identical to the value of e in the current state. The operator K is the epistemic knowledge operator. $K\phi$ holds if ϕ holds in any state equivalent to the current state. In our setting, two states are considered equivalent if the same sequence of outputs has been generated before reaching them. The operator U is the standard (strong) until operator. The formula $\phi U \psi$ holds if ψ holds in a future state and ϕ holds until reaching that state.

Various connectives are definable in $\mathcal{L}_{KU}$ including standard derived boolean operators such as $\vee$ and $\rightarrow$, the truth constants *tt* and *ff*, universal $\forall x$ and existential $\exists x$ quantifiers over the finite set of values, the epistemic possibility operator

$L\phi$ meaning that ϕ holds for at least one epistemically equivalent state, the future operator $F\phi$ requiring ϕ to eventually hold in the future, the "always" operator $G\phi$ meaning that ϕ holds in any future state, and the weak until $\phi W \psi$ which does not require ψ to eventually hold. In the remainder of the paper, we use the above connectives as syntactic sugar with the following definitions.

Definition 2.3.2 (Syntactic sugar $\forall$, $\exists$, L, F, G and W)

$$\forall x.\, \phi = \bigwedge_{v \in Val} \phi[v/x] \qquad \exists x.\, \phi = \bigvee_{v \in Val} \phi[v/x]$$

$$L\phi = \neg K(\neg\phi) \qquad F\phi = tt\, U \phi \qquad G\phi = \neg(F\neg\phi)$$

$$\phi W \psi = (\phi U \psi) \vee G\phi$$

Since there is no input statement in the programming language, the only way for secrets to enter a computation is through the initial state. This, and also the lack of past-time temporal connectives which would in a more general setting of reactive programs be a natural device to record past inputs, explains the purpose of the initial state predicate $init_x(e)$ which plays a critical role in capturing what is known "now" of the initial store. It has to be noted that if e is independent from the current state then, as the initial value of x does not change over time, the majority of temporal variations of $init_x(e)$ do not change its semantics as long as the computation has not terminated yet ($init_x(e) = F init_x(e) = G init_x(e) = \phi U init_x(e)$).

Noteworthy, also, is that outputs are not reflected in the syntax of the logic by corresponding operators or constants. The reason is that output events are of no intrinsic interest to us; they are relevant only in terms of their effect on observer knowledge, of which states are considered equivalent with regard to operators K and L.

Definition 2.3.3 (Satisfaction)
Fig. 2.2 defines the satisfaction relation $\mathcal{M}, (\pi, i) \models \phi$ between points in a model $\mathcal{M}$ and formulas. If the model $\mathcal{M}$ is clear from the context, we write $(\pi, i) \models \phi$ or $\pi, i \models \phi$ for $\mathcal{M}, (\pi, i) \models \phi$. Satisfaction relative to model $\mathcal{M}$ or program P is:

$$\mathcal{M} \models \phi \qquad \text{iff } \forall \pi \in \mathcal{M},\ \mathcal{M}, (\pi, 0) \models \phi$$
$$P \models \phi \qquad \text{iff } \mathcal{M}(P) \models \phi$$

In terms of epochs the formula $K\phi$ expresses that ϕ holds for all points in the current epoch; and, dually, $L\phi$ expresses that ϕ holds for at least one point in the current epoch, or in other words, that the observer is unable to rule out $\neg\phi$ on the basis of the outputs received so far.

Example 2.3.1 (Basic example) *If the point (π, i) satisfies the formula $G(x = 5)$ then, in all future execution points of π, variable x has value 5. If (π, i) satisfies*

$$
\begin{array}{ll}
\mathcal{M},(\pi,i) \models e_1 = e_2 & \text{iff } \sigma(\pi,i)(e_1) = \sigma(\pi,i)(e_2) \\
\mathcal{M},(\pi,i) \models \mathit{init}_x(e) & \text{iff } \sigma(\pi,0)(x) = \sigma(\pi,i)(e) \\
\mathcal{M},(\pi,i) \models \phi \wedge \psi & \text{iff } (\pi,i) \models \phi \text{ and } (\pi,i) \models \psi \\
\mathcal{M},(\pi,i) \models \neg\phi & \text{iff } (\pi,i) \not\models \phi \\
\mathcal{M},(\pi,i) \models K\phi & \text{iff } \forall \pi' \in \mathcal{M}, \forall (\pi',i') \in \pi' \text{ such that} \\
 & \quad \mathit{trace}(\pi,i) = \mathit{trace}(\pi',i'), (\pi',i') \models \phi \\
\mathcal{M},(\pi,i) \models \phi U \psi & \text{iff } \exists j : i \leq j \leq \mathit{len}(\pi) \text{ such that} \\
 & \quad (\pi,j) \models \psi \text{ and } \forall k : i \leq k < j,\ (\pi,k) \models \phi
\end{array}
$$

Figure 2.2: Formulas satisfaction at execution point

the formula $F(K\phi)$ then there exists a point (π, j) (with $j \geq i$) for which ϕ holds for all points (π', j') (including (π, j)) having the same trace as (π, j) ($\mathit{trace}(\pi, j) = \mathit{trace}(\pi', j')$, i.e. execution π' after j' steps has generated the same output sequence as execution π after j steps). Combining both previous formulas, if (π, i) satisfies the formula $FKG(x = 5)$ then there exists a trace τ of a future point (π, i) for which x equals 5 in every future point of any point having trace τ.

Example 2.3.2 (It is always possible to lose) *At the program level, if the formula $GLF(\mathit{lost} = tt)$ for program P then, for all potential traces τ of P, there exists an execution of P which at one point has generated the trace τ and for which lost will be equal to tt at some point in the future. In other words, if the initial state of an execution of P is unknown, whatever output sequence is observed, it is impossible to rule out the fact that losing in the future is still possible.*

Example 2.3.3 (Eventually, the initial value is deducible) *Still at the program level, if $\exists v.\ FK\mathit{init}_x(v)$ holds for program P then for all executions π of P there exists a value v and a point (π, i) which generates a trace τ for which, for any execution π' of P, all points (π', i') generating the same trace τ (including (π, i)) are such that the initial value of x is v. In other words, any execution of P will, at some point, have generated an output sequence from which it is possible to deduce the initial value of x.*

2.3.1 Relation to Standard Models of Knowledge

Kripke structures are commonly used to give semantics to modal logics, and hence by extension to epistemic logics as well [112]. A Kripke structure (for a single agent) is a triple $(S, \mathcal{T}, \mathcal{K})$ where S is a set of states, $\mathcal{T}$ is a valuation assigning to each atomic proposition a predicate on S, and $\mathcal{K}$ is a binary accessibility relation on states such that $(s_1, s_2) \in \mathcal{K}$ if from the observations made by the observer while

in state s_1, it is equally possible to be in state s_2. For a given model $\mathcal{M}$, let $S_\mathcal{M}$ be the set of all the execution points (π, i) of the executions π of $\mathcal{M}$; let $\mathcal{T}_\mathcal{M}$ be the function taking each atomic proposition of the shape "$e_1 = e_2$" or "$\mathit{init}_x(e)$" to the set of points for which the proposition holds according to Def. 2.3.3; and finally, let $\mathcal{K}_\mathcal{M}$ be the binary relation $\sim$ defined at the end of Sect. 2.2. Then $(S_\mathcal{M}, \mathcal{T}_\mathcal{M}, \mathcal{K}_\mathcal{M})$ is a Kripke structure for which the standard definitions of the knowledge operators have the same semantics as the one provided in Def. 2.3.3.

Interpreted systems are a refinement of Kripke structures used to define the semantics of epistemic logics [112, 187] in terms of multi-agent systems. Roughly, an interpreted system is a pair $(\mathcal{R}, \mathcal{T})$, where $\mathcal{R}$ is a set of runs r as functions from time to global states. A global state is a tuple composed of an environment state and one state for every agent in the system. Similarly as in the case of Kripke structures, $\mathcal{T}$ is a function stating if a state formula holds on a given global state. For a given model $\mathcal{M}$, let $\mathcal{R}_\mathcal{M}$ be the set of runs r_π such that $\pi \in \mathcal{M}$ and $r(i)$ is the pair composed of the environment state $\mathit{trunc}(\pi, i)$ with actions removed and the agent/attacker state $\mathit{trace}(\pi, i)$. Let $\mathcal{T}$ be defined for formulas of the shape "$e_1 = e_2$" or "$\mathit{init}_x(e)$" according to Def. 2.3.3, as a predicate on global states. The semantics of the knowledge operators provided in Def. 2.3.3 is equivalent to their standard semantics over the interpreted system $(\mathcal{R}_\mathcal{M}, \mathcal{T}_\mathcal{M})$.

2.4 Noninterference

We now discuss how the logic applies to information flow security properties, adapted to the present setting of output-only imperative programs. We first consider the concept of noninterference [122]. In a language-based setting and considering a two-level security lattice only, noninterference in a relational (initial-final state) setting requires that no information about initial values of high identifiers (which we want to protect) can flow to final values of low identifiers (which the attacker can observe). This condition is easily adapted to the present setting of output-only programs by instead prohibiting high flow to the public outputs.

Write $\sigma_1 \approx_{\vec{x}} \sigma_2$ if the two stores σ_1 and σ_2 are equivalent with regard to a set of identifiers $\vec{x}$, i.e. $\forall x \in \vec{x}.\, \sigma_1(x) = \sigma_2(x)$. Fix now a set of low identifiers $\vec{l}$, and let $\vec{h}$ be its complement, the high identifiers.

Definition 2.4.1 (ONI)
A program P satisfies output-only noninterference *iff:*

$$\forall \pi_1, \pi_2 \in \mathcal{M}(P).\, \sigma(\pi_1, 0) \approx_{\vec{l}} \sigma(\pi_2, 0) \Rightarrow \mathit{trace}(\pi_1) = \mathit{trace}(\pi_2)$$

Intuitively, the definition states that there is no information flowing from $\vec{h}$ to the attacker if for any maximal execution having trace τ, all maximal executions started with the same values for $\vec{l}$ produce the same trace. In other words, all initial secret values ($\vec{h}$) might have given rise to the output sequence that an attacker is observing. It is worth noting that this definition subsumes standard noninterference. Indeed,

we only need to modify program P by outputting the values of low identifiers ($\vec{l}$) whenever they are observable. Termination sensitivity can also be added by a final dummy output. We now show how ONI can be encoded in our epistemic framework.

Definition 2.4.2 (ESP)

$$\mathsf{ESP} \stackrel{def}{=} \forall \vec{v}.\, (init_{\vec{l}}(\vec{v}) \rightarrow \forall \vec{u}.\, L(init_{\vec{l}}(\vec{v}) \wedge init_{\vec{h}}(\vec{u})))$$

The formula ESP is satisfied at a given execution point if every initial secret is possible among the execution points having the same trace and initial public values. In our epistemic framework, we claim that a program does not reveal any secret if all its execution points satisfy ESP, i.e. every initial secret is possible for every trace and public inputs generating such trace.

Definition 2.4.3 (AK)
A program P satisfies absence of knowledge *iff:*

$$P \models\ G(\mathsf{ESP})$$

We first give some examples to show how the logic applies to programs wrt. standard noninterference and afterwards prove the equivalence of the above definitions.

Example 2.4.1 *Let $P ::= x := y; \mathtt{out}(y)$ be a program over booleans with $x \in \vec{h}, y \in \vec{l}$. Then P satisfies ONI since the initial value of y never changes. We show that P satisfies AK. Consider a model $\mathcal{M}$ associated with program P where the store is a pair (x, y). Then*

$$\mathcal{M} ::= \begin{cases} \pi_1 = (tt, tt) \rightarrow (tt, tt) \stackrel{tt}{\rightarrow} (tt, tt) \\ \pi_3 = (tt, ff) \rightarrow (ff, ff) \stackrel{ff}{\rightarrow} (ff, ff) \\ \pi_2 = (ff, ff) \rightarrow (ff, ff) \stackrel{ff}{\rightarrow} (ff, ff) \\ \pi_4 = (ff, tt) \rightarrow (tt, tt) \stackrel{tt}{\rightarrow} (tt, tt) \end{cases}$$

One can verify, by case analysis, that $\mathcal{M} \models G(\mathsf{ESP})$. Consider for instance π_4. Then $v = tt$ and $\pi_4, i \models init_y(v)$ holds for all $0 \leq i \leq 2$. We show that $\pi_4, i \models \forall u.L(init_y(v) \wedge init_x(u))$ for all i. For $i \in \{0, 1\}$, $trace(\pi_4, i) = \epsilon$, so we can find $(\pi_1, 0)$ and $(\pi_2, 0)$ if $u = tt$ and $u = ff$, respectively. If $i = 2$ and $u = tt$, then $(\pi_1, 2)$ has the same trace and initial value; otherwise, if $u = ff$, we pick $(\pi_4, 2)$. Similarly, the condition holds for other cases.
Let now $P ::= x := y; \mathtt{out}(y)$ with $x \in \vec{l}, y \in \vec{h}$. Then, P falsifies ONI since we output the secret value y to public output. We show for model $\mathcal{M}$ that $\mathcal{M} \not\models G\, \forall v.(init_x(v) \rightarrow \forall u.L(init_x(v) \wedge init_y(u)))$ i.e. $\exists \pi. \exists i. \exists v. init_x(v) \wedge \exists u. \forall \pi'.$ $\forall i'. trace(\pi, i) = trace(\pi', i')$ then $\pi', i' \not\models (init_x(v) \wedge init_y(u)))$. In particular, π_3 is a counterexample. Set $v = tt$ and $u = tt$; the only executions having the same trace as π_3 are π_2 and π_3. However, $\sigma(\pi_2, 0)(x) = ff \neq v$ and $\sigma(\pi_3, 0)(y) = ff \neq u$.

Lemma 2.4.1 (Initial values stability) *For all vectors of values $\vec{v}$ and identifiers $\vec{x}$:*

$$\pi, 0 \models init_{\vec{x}}(\vec{v}) \text{ implies } \forall(\pi, i) \in \pi : \ \pi, i \models init_{\vec{x}}(\vec{v})$$

PROOF. *Immediate. By definition of satisfaction relation $\pi, i \models init_{\vec{x}}(\vec{v})$ iff $\sigma(\pi, 0)(\vec{x}) = \vec{v}$.* □

Proposition 2.4.1 (Equivalence of ONI and AK) *For all programs P:*

$$P \models ONI \ \textit{iff} \ P \models AK$$

PROOF. ($\Rightarrow$) *Assume P satisfies ONI. By definition, given π_1, then for all π_2. $\sigma(\pi_1, 0) \approx_{\vec{l}} \sigma(\pi_2, 0)$, $trace(\pi_1) = trace(\pi_2)$. In particular any two equal traces have equal prefix traces of same length. We show that $\pi \in \mathcal{M}$. $\pi, 0 \models \ G\ \forall\vec{v}.(init_{\vec{l}}(\vec{v}) \rightarrow \forall\vec{u}.L(init_{\vec{l}}(\vec{v}) \wedge init_{\vec{h}}(\vec{u})))$. Pick any $\pi \in \mathcal{M}$ and $\vec{v} \in Val$; then we show for all $0 \leq i \leq len(\pi)$. $\pi, i \models (init_{\vec{l}}(v) \rightarrow \forall\vec{u}.L(init_{\vec{l}}(v) \wedge init_{\vec{h}}(\vec{u})))$. Namely, assume $\pi, i \models init_{\vec{l}}(\vec{v})$ then for any $\vec{u} \in Val$ there exists π', i'. $trace(\pi, i) = trace(\pi', i') \wedge (\pi', 0) \models init_{\vec{l}}(\vec{v}) \wedge init_{\vec{h}}(\vec{u})$. Let now $\mathcal{M}_a \subseteq \mathcal{M}$ be such that $\forall\pi \in \mathcal{M}_a$. $\sigma(\pi, 0)(l) = a$. Then $\mathcal{M} = \bigcup_{a \in Val} \mathcal{M}_a$. By ONI condition, for all $\pi \in \mathcal{M}_a$. $trace(\pi) = \tau$ for some trace τ and any initial $\vec{h}$. Then, using Lemma 2.4.1 and chopping off execution π we get the result for all (π, i). The same argument can be used for any $\mathcal{M}_a$, so we are done.*
($\Leftarrow$) *Suppose now $\forall\pi \in \mathcal{M}$. $\pi, 0 \models \ G\ \forall\vec{v}.(init_{\vec{l}}(v) \rightarrow \forall\vec{u}.L(init_{\vec{l}}(v) \wedge init_{\vec{h}}(\vec{u})))$. We show ONI holds. By hypothesis, pick $\pi \in \mathcal{M}$ with $\sigma(\pi, 0)(l) = v$, then we show that for all π' such that $\sigma(\pi', 0)(l) = v$, $trace(\pi) = trace(\pi')$. By hypothesis, given π, in particular it is always possible to find π' with same initial values $\vec{v}$, for any $\vec{u}$ having the same trace.* □

Example 2.4.2 *Let P be a program manipulating two private variables h_1, h_2 over boolean domain.*

$$P ::= \ \texttt{if}\ h_1\ \texttt{then}\ \texttt{out}(\neg h_2)\ \texttt{else}\ \texttt{out}(h_2)$$

The program is not secure since it reveals whether the secrets are equal or not i.e. $h_1 = h_2$. In fact, for all input states where $h_1 = h_2$ i.e. $(tt, tt), (ff, ff)$, P outputs ff, otherwise it outputs tt and this is captured by Def. 2.4.3.

On the other hand, we will see in the following section that if one agrees to declassifies $\phi := h_1 = h_2$ then Def. 2.5.3 will deem the program secure.

2.5 Declassification: What

Noninterference guarantees an end-to-end confidentiality policy, namely as soon as a program conveys 1 bit of secret information, it is ruled out by the condition. In

real applications this policy turns out to be restrictive, as in many scenarios partial information leakage is considered admissible. Declassification policies handle those acceptable, or even desired, information leakages [202]. For example, a customer may be allowed to access a scientific article (secret data) once she has paid the registration fee to some on line provider. In this case, an intentional release of secret information is needed. Declassification has been recognized as one of the main challenges in information flow security [197]. The main concern is to prove that declassification is safe and the attacker is unable to compromise the release mechanism and disclose more sensitive information than stated in the policy. Many authors have addressed the problem from different points [87, 198, 160, 28, 50, 41]. In particular, in [202], the authors present a classification of different flavors of declassification. In this section and the following ones, we show how our temporal epistemic framework captures in an elegant way those dimensions.

One way of modeling declassification is by means of a predicate ϕ over initial values which expresses the property one intends to declassify. In that case, one has to make sure that states having the same property ϕ can not be distinguished by the attacker. This idea originates from *selective dependency* [87] and corresponds to the *What* dimension [202]. In particular, the programmer should specify a global declassification policy ϕ and the enforcement mechanism has to ensure that no information other than what is specified in the policy can be disclosed by the attacker. For example, the information system of a company can release the average salary of an employee, but it shouldn't be possible to reveal, for instance, the salary of a certain employee. Let $\sigma_1 \approx_\phi \sigma_2$ denote equivalent states according to the declassification policy ϕ i.e. $\sigma_1(\phi) = \sigma_2(\phi)$.

Definition 2.5.1 (NID)
Let ϕ be a global declassification policy. A program P satisfies noninterference modulo declassification ϕ *iff:*

$$\forall \pi_1, \pi_2 \in \mathcal{M}(P).\ (\sigma(\pi_1, 0) \approx_{\vec{l}} \sigma(\pi_2, 0) \wedge \sigma(\pi_1, 0) \approx_\phi \sigma(\pi_2, 0)) \Rightarrow \mathit{trace}(\pi_1) = \mathit{trace}(\pi_2)$$

The definition of NID specifies that any initial state having the same public values and agreeing on ϕ should produce the same output trace.

Let us now see how global declassification policies can be expressed in our model. We first introduce the formula ESPM. An execution point satisfies $\mathsf{ESPM}(\Phi)$ where Φ is a set of declassification policies iff, among the other execution points having the same trace and initial public values, every initial secret agreeing on Φ is possible.

Definition 2.5.2 (ESPM)

$$\mathsf{ESPM}(\Phi) \stackrel{def}{=} \quad \forall \vec{v}_1.\, \forall \vec{u}_1.\, init_{\vec{l}}(\vec{v}_1) \wedge init_{\vec{h}}(\vec{u}_1) \rightarrow \forall \vec{u}_2.\, (\bigwedge_{\phi \in \Phi} \phi(\vec{v}_1, \vec{u}_1) = \phi(\vec{v}_1, \vec{u}_2)) \rightarrow L(init_{\vec{l}}(\vec{v}_1) \wedge init_{\vec{h}}(\vec{u}_2))$$

Proposition 2.5.1 (Equivalence of ESP and ESPM($\emptyset$)) *For all execution points* (π, i)*:*

$$(\pi, i) \models \mathsf{ESP} \quad \textit{iff} \quad (\pi, i) \models \mathsf{ESPM}(\emptyset)$$

PROOF. *This proposition follows directly from the fact that if* Φ *is empty then* $\bigwedge_{\phi \in \Phi}$ *is vacuously true and* $init_{\vec{h}}(\vec{u}_1)$ *holds for at least one vector of values* $\vec{u}_1$*.* □

Proposition 2.5.2 (Monotonicity of ESPM) *For all execution points* (π, i) *and sets of declassifications* Φ *and* Ψ*:*

$$(\pi, i) \models \mathsf{ESPM}(\Phi) \quad \textit{implies} \quad (\pi, i) \models \mathsf{ESPM}(\Phi \cup \Psi)$$

PROOF. *This proposition follows trivially from the second implication in the formula of* ESPM*. Whenever the left part of the implication* $\bigwedge_{\phi \in \Phi \cup \Psi}$ *holds then* $\bigwedge_{\phi \in \Phi}$ *also holds; and the right part of the implication is the same in both cases, so if the L formula holds with* Φ *it still holds with* $\Phi \cup \Psi$*.* □

Corollary 2.5.1 (ESP **subsumes** ESPM) *For all execution points* (π, i) *and sets of declassifications* Φ*:*

$$(\pi, i) \models \mathsf{ESP} \quad \textit{implies} \quad (\pi, i) \models \mathsf{ESPM}(\Phi)$$

PROOF. *This is a direct corollary of Prop. 2.5.1 and 2.5.2.* □

Definition 2.5.3 (AKD)
Let ϕ *be a global declassification policy. A program* P *satisfies* absence of knowledge modulo declassification ϕ *iff:*

$$P \models G(\mathsf{ESPM}(\{\phi\}))$$

Figure 2.3 illustrates the intuition behind our security condition. The graphic presents the knowledge about initial secrets that an attacker gains by observing a certain trace $\tau = o_1\, o_2\, o_3$ as function of time elapsed from the beginning of computation. The black solid line shows the evolution of attacker knowledge at each output point and in particular how it can possibly increase in each epoch. Initially the

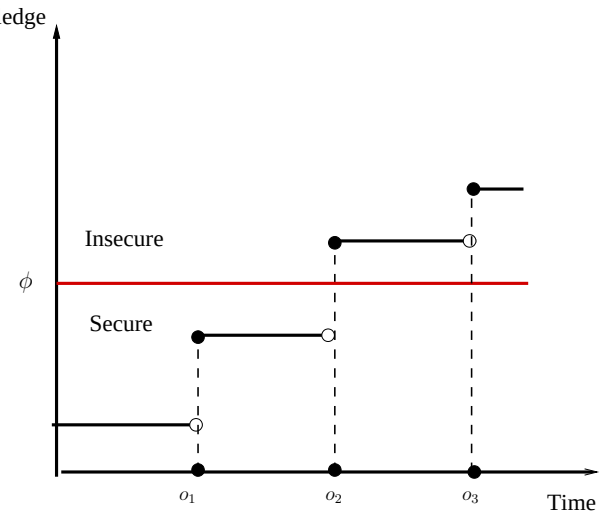


Figure 2.3: Knowledge and Declassification

attacker has knowledge about public identifiers. On the other hand the red dotted line shows the global declassification policy represented by a predicate ϕ. As long as the solid line remains below the dotted line the declassification is safe, namely the attacker knowledge is smaller than the information released intentionally prior to program execution. In this case, one can see that after the second observation point o_2 the attacker learns more than the policy allows, thus the program becomes insecure.

Proposition 2.5.3 (Equivalence of NID and AKD) *For all programs P:*

$$P \models NID \;\; \textit{iff} \;\; P \models AKD$$

PROOF. *The proof is similar to the one for Prop. 2.4.1.* □

It is worth noting that if the declassification policy states "*No secret information can be leaked*", then the property becomes $\phi = tt$ and AKD will correspond to AK. We illustrate the above condition by means of an example.

Example 2.5.1 *Consider the program P with $h \in \vec{h}$.*

$$P ::= \texttt{if}\ (h = 0)\ \texttt{then}\ \texttt{out}(1)\ \texttt{else}\ \texttt{out}(2)$$

One can spot an implicit flow due to dependence on a conditional on secret h. Let $\mathcal{M}$ be a model of P. To falsify Def. 2.4.3, pick π such that $\sigma(\pi, 0)(l) = \sigma(\pi, 0)(h) = 0$.

Then, pick π' such that $\sigma(\pi', 0)(l) = 0$ and $\sigma(\pi', 0)(h) \neq 0$. It is easy to see that $trace(\pi) \neq trace(\pi')$. Suppose now we declassify the zeroness of h i.e. $\phi := (h = 0)$. All executions originating from $h = 0$ produce the same trace i.e. output 1. On the other hand, all executions originating from $\neg\phi := (h \neq 0)$ also produce the same trace, i.e. output 2. Hence, the program is secure. It is worth to noting how Def. 2.5.3 rules out programs that reveal more than what is allowed by the declassification policy. Suppose we want to declassify the sign of identifier h, namely $\phi := (h \geq 0)$. Then, P becomes insecure since the attacker is now able to distinguish between values having the same property ϕ. In particular let $h_1 = 0$ and $h_2 = 1$, so $\phi(h_1) = \phi(h_2)$. In that case P outputs 1 and 2, respectively, so it is deemed insecure.

Abstract Non-Interference Abstract Non-Interference (ANI) is an abstract interpretation based approach for modeling and certifying information flow properties [119]. This framework characterizes different qualitative aspects related to global declassification policies and attacker observational power. In particular, using the notion of abstract domain, the authors give an extensional model of what an attacker is allowed to see of public data (attacker power) and of what she is allowed to disclose of secret data (declassification). For example, let P be a program with $l \in \vec{l}, h \in \vec{h}$.

$$P ::= \texttt{if } (h \geq 0) \texttt{ then } l := 2l * h \texttt{ else } l := 2l * h + 1$$

Clearly, there is an direct flow to public identifier l which conveys the value of secret h. However, if one is interested in releasing only the sign of secret identifier h in input and considers a weaker attacker who is able to observe only the parity of identifier l in output then P will be secure. Indeed, fix the initial value of low identifier l and consider initial values of h in input having the same sign, say $h < 0$. It can be easily seen that the final value of l will have the same parity; in this case it will correspond to an odd value. This definition is called *Narrow ANI via allowing* [163]. Let η, ϕ, ρ be the abstract domains for public input, declassified private input and public output, respectively.

Definition 2.5.4 (NANI)
A program P satisfies Narrow ANI, *$(\eta)P(\phi \Rightarrow \rho)$, iff:*

$$\forall l_1, l_2 \in \vec{l}, \forall h_1, h_2 \in \vec{h} : \eta(l_1) = \eta(l_2) \wedge \phi(h_1) = \phi(h_2) \Rightarrow \rho([\![P]\!](h_1, l_1)) = \rho([\![P]\!](h_2, l_2))$$

Basically it states that for any initial public values having property η and for any private initial values having property ϕ, the result of the computation has property ρ over public outputs. In particular the previous example corresponds to checking $(Id)P(Sign \Rightarrow Par)$.
There is a nice relation between NANI and our epistemic framework. One can

look at the abstractions over public input domain and public output domain as abstractions over channels receiving and releasing these values, respectively. More concretely, suppose one wants to check NANI for $(\eta)P(\phi \Rightarrow \rho)$. In order to model the attacker power in output we can use the output actions $\mathtt{out}(e)$ and check the following formula wrt. a model $\mathcal{M}$ of the program $P; \mathtt{out}(\rho(l))$. Given a pair $(\vec{u}, \vec{v})$ we denote by *fst* and *snd*, respectively, the first and the second component of such a pair.

Definition 2.5.5 (AAK)
A program P satisfies abstract absence of knowledge *w.r.t. abstractions ρ, η and ϕ iff:*

$$P \,;\, \mathtt{out}(\rho(\vec{l})) \models G(\mathsf{ESPM}(\{\eta \circ \mathit{fst}, \phi \circ \mathit{snd}\}))$$

On the other hand, the public input abstraction η deserves some explanation. It can happen that Def. 2.5.5 fails because the attacker is able to distinguish two input states having the same property η. Consider a model $\mathcal{M}$ of the program $P ::= l := 2l * h^2; \mathtt{out}(\mathit{Sign}(l))$ where $\eta = \mathit{Par}$ and $\phi = \mathit{Id}$. Let π be a maximal execution originating from initial state σ such that $\sigma(\pi, 0)(l) = 2$ and $\sigma(\pi, 0)(h) = 1$. Then one can find another maximal execution π' such that $\sigma(\pi', 0)(l) = -2$ and $\sigma(\pi', 0)(h) = 1$. Clearly $\mathit{Par}(\sigma(\pi, 0)(l)) = \mathit{Par}(\sigma(\pi', 0)(l))$ and $\phi = tt$, while the sign of the outputs are different i.e. $\mathit{Sign}(4) \neq \mathit{Sign}(-4)$. In [119] this is called *deceptive* flow, since it only depends on variations of public inputs. However, if one interprets the public input abstraction η as secret knowledge that should not be controlled or disclosed to the attacker then it is reasonable to rule out the program above. Indeed, here the attacker is disclosing a property stronger than *Par* since she observes variations of the sign for inputs of even parity.
We now show the equivalence of these definitions and postpone a further investigation of relation to abstract non-interference as future work.

Proposition 2.5.4 (Equivalence of NANI and AAK) *For all programs P:*

$$P \models \mathit{NANI} \;\; \mathit{iff} \;\; P \models \mathit{AAK}$$

Proof. *It is enough to observe that the abstract domain ρ in NANI can be considered as a predicate over public output states. In that case the output action in AAK models the same property.* □

We conclude this section by discussing an interesting example.

Example 2.5.2 *Let P be a program that manipulates a secret variable $h \in \vec{h}$, initially known to range over non-negative numbers up to some constant max. We*

express this fact by a declassification policy $\phi = 0 \leq h \leq max$. Then P is secure since it outputs the same sequence of numbers in every run.

$$P ::= \left[\begin{array}{l} x := 0; \\ \texttt{while}\ (x < h)\ \texttt{do}\ \texttt{out}(x); x++; \\ \texttt{while}\ (x < max)\ \texttt{do}\ \texttt{out}(x); x++; \end{array}\right.$$

Program P satisfies Def 2.5.3. Too see this, consider a model $\mathcal{M}$ of P, a maximal execution π originating from $\sigma_0 = (max_0, x_0, h_0)$ and any point i. $0 \leq i \leq len(\pi)$. Assume $\phi(h_0)$ holds, then for all values h_i such that $\phi(h_i)$, it is possible to find an execution π' originating from (max_0, x_0, h_i) and a point i' such that $trace(\pi, i) = trace(\pi', i')$. In fact, all executions produce a increasing trace of numbers of length at most max_0. If $\phi(h_0)$ does not hold then all executions produce the empty trace.

2.6 Declassification: Where

Another well-studied form of declassification regards where in the system sensitive information can be released. In our framework, the only way to leak secret information is by means of output operations. In particular, any flow of information from a high identifier h to a low identifier l is perfectly fine as long as secret data is not being output. It is irrelevant at which point of a certain epoch the declassification occurs. For this reason, assume that declassification takes place together with the output actions. We model the release points in the code by special boolean flags r_e initially *false* and once set to *true* the program can release the value of expression e. Moreover, the flag can no more be updated once it is set to true. Assume we are given a set of release points interspersed in the program, say $\mathcal{R}_p = \{r_{e_1}, \cdots, r_{e_n}\}$, and the corresponding release expressions $\mathcal{R} = \{e_1, \cdots, e_n\}$ then the goal is to check whether program P leaks more information that what the programmer has already allowed to be disclosed by means of the release points encountered so far. It is worth recalling that our model intends to protect the initial value of secret data, not the current ones. This objective is in line with most other work on non-interference. Let $\mathcal{P}(\mathcal{R})$ be the power set of $\mathcal{R}$ and $\bar{\mathcal{E}}$ be the complement of $\mathcal{E}$ in $\mathcal{R}$. The formula expressing the absence of attacker knowledge is given next.

Definition 2.6.1 (AKR)
Let $\{r_{e_1}, \cdots, r_{e_n}\}$ be the boolean variables, initially false, serving as flags for the release policy $\mathcal{R}$. A program P satisfies absence of knowledge modulo release $\mathcal{R}$ *iff:*

$$P \models G \bigvee_{\mathcal{E} \in \mathcal{P}(\mathcal{R})} \Big(\mathsf{ESPM}(\mathcal{E}) \wedge \bigwedge_{e_i \in \mathcal{E}} r_{e_i} \wedge \bigwedge_{e_j \in \bar{\mathcal{E}}} \neg r_{e_j} \Big)$$

Note that the conditions above are mutually exclusive with respect to release points, namely given π and i, only one formula in the disjunction holds and that corresponds to the one with release points set to true in execution $trunc(\pi, i)$.

Example 2.6.1 *Consider program P with $h_1, h_2 \in \vec{h}$ and $l \in \vec{l}$.*

$$l := h_1\,;\ r_{h_1} := tt\,;\ \texttt{out}(l);\ l := h_2\,;\ r_{h_2} := tt\,;\ \texttt{out}(l);$$

Stores are vectors (l, h_1, h_2) and $\vec{h}$ is the high store (h_1, h_2). Intuitively P is secure since the value of a secret is always declassified before being output. Pick $\pi \in \mathcal{M}(P)$. We show that Def. 2.6.1 holds for $(\pi, 0)$. Initially $\mathcal{E} = \emptyset$ is the only candidate such that $\bigwedge_{e_i \in \mathcal{E}} r_{e_i} \wedge \bigwedge_{e_j \in \bar{\mathcal{E}}} \neg r_{e_j}$. It remains to prove that $\pi, 0 \models \mathsf{ESPM}(\emptyset)$. This trivially holds until the first release point as the trace of any execution up to this point is empty and any execution generates an empty trace at some point. Then, we move on to $(\pi, 2)$ which is the first execution point after setting the first release flag. At this point, $\mathsf{ESPM}(\{h_1\})$ is required to hold. For the same reason as above, $\mathsf{ESPM}(\emptyset)$ holds and by Prop. 2.5.2 $\mathsf{ESPM}(\{h_1\})$ also holds. The trace of $(\pi, 3)$ is "$\mathtt{h}_1$", where $\mathtt{h}_1$ is the initial value of h_1, and $\mathsf{ESPM}(\{h_1\})$ is still the formula required to hold. Among all the execution points whose trace is $\mathtt{h}_1$ and whose execution has started with the same initial values for l and h_1, there is at least one point whose execution has started with $h_2 = \mathtt{h}_2$ for any $\mathtt{h}_2$. Hence, $(\pi, 3)$ satisfies $\mathsf{ESPM}(\{h_1\})$. Similarly, $(\pi, 4) \models \mathsf{ESPM}(\{h_1\})$, $(\pi, 5) \models \mathsf{ESPM}(\{h_1, h_2\})$ and $(\pi, 6) \models \mathsf{ESPM}(\{h_1, h_2\})$. Hence, P satisfies AKR.

We now show how Def. 2.6.1 relates to a similar security condition called *gradual release* [28]. Although gradual release considers a slightly different computational model, the basic idea is that the attacker knowledge is constant between release points. In the same spirit, we compute the attacker knowledge for a given trace and compare it with the information released over that trace. In particular, if the attacker knowledge is greater than what has been declassified so far, there is an insecure leakage. Given a program P, an initial store σ_0 and a trace τ originating from that store, we define the knowledge over the trace $\mathcal{K}(P, \sigma_0, \tau)$ as the set of initial stores that could have led to that trace.

$$\mathcal{K}(P, \sigma_0, \tau) = \{\sigma(\pi, 0) \mid \exists(\pi, i) : \sigma(\pi, 0) \approx_{\vec{l}} \sigma_0 \wedge \mathit{trace}(\pi, i) = \tau\}$$

As pointed out by Askarov and Sabelfeld [28], this set corresponds to the uncertainty of an attacker observing trace τ.

When reaching a point whose trace is τ and execution started in σ_0, a certain number of release point r_ϕ have been executed. Let $\mathcal{D}_{\sigma_0,\tau}$ be the set of common release points that have been executed when reaching any point whose trace is τ and execution started in σ_0 and $\Phi_{\sigma_0,\tau} = \{\phi \mid r_\phi \in \mathcal{D}_{\sigma_0,\tau}\}$. Moreover, let $\mathcal{R}(P, \sigma_0, \tau)$ be the maximum knowledge authorized, or minimum uncertainty required, at a point whose trace is τ for an execution started with the value store σ_0.

$$\mathcal{R}(P, \sigma_0, \tau) = \{\sigma \mid \sigma \approx_{\vec{l}} \sigma_0 \wedge \bigwedge_{\phi \in \Phi_{\sigma_0,\tau}} \sigma_0(\phi) = \sigma(\phi)\}$$

Then, a program is secure if the information disclosed by observing a given trace is less than the information released over that trace; or if the required uncertainty is a subset of the attacker uncertainty.

Definition 2.6.2 (ER)
A program P satisfies epistemic release *iff:*

$$\forall \sigma_0, \tau : \ \mathcal{R}(P, \sigma_0, \tau) \subseteq \mathcal{K}(P, \sigma_0, \tau)$$

Example 2.6.2 *Consider the program in Example 2.6.1 over a boolean domain and (l, h_1, h_2) a triple corresponding to a store. Take $\sigma_0(l) = tt$. Then, for the empty trace ϵ, we have $\mathcal{K}(P, \sigma_0, \epsilon) = \mathcal{R}(P, \sigma_0, \epsilon) = \{(tt, \_, \_)\}$. Now we pick $\tau = tt$ and $\mathcal{K}(P, \sigma_0, tt) = \mathcal{R}(P, \sigma_0, tt) = \{(tt, tt, \_)\}$ since we release h_1. Proceeding in this way it is easy to prove that P satisfies ER. Suppose that we don't release h_1 at the first output. Then we have $\mathcal{R}(P, \sigma_0, tt) = \{(tt, \_, \_)\}$ which is clearly not contained in $\mathcal{K}(P, \sigma_0, tt)$.*

Proposition 2.6.1 (Equivalence of AKR and ER) *For all programs P:*

$$P \models AKR \ \ \textit{iff} \ \ P \models ER$$

Proof. ($\Rightarrow$) *Assume $P \models AKR$. Let $\pi \in \mathcal{M}(P)$. We show that for all prefixes τ of $trace(\pi), \mathcal{R}(P, \sigma(\pi, 0), \tau) \subseteq \mathcal{K}(P, \sigma(\pi, 0), \tau)$. Consider (π, i) such that $trace(\pi, i) = \tau$ and release points $r_{\phi_1}, \cdots, r_{\phi_k}$ being active. By Def. 2.6.1, $\pi, i \models \mathsf{ESPM}(\mathcal{E})$ where $\mathcal{E} = \{\phi_1, \cdots, \phi_k\}$. Basically, it says that for all $(\pi', 0)$ such that $\sigma(\pi, 0) \approx_{\vec{l}} \sigma(\pi', 0)$ and $\bigwedge_{\phi \in \mathcal{E}} \sigma(\pi, 0)(\phi) = \sigma(\pi', 0)(\phi)$ (i.e. $(\pi', 0) \in \mathcal{R}(P, \sigma_0, \tau)$), there exists (π', i') such that $trace(\pi', i') = \tau$ (i.e. $(\pi', 0) \in \mathcal{K}(P, \sigma_0, \tau)$). This is exactly ER.*
($\Leftarrow$) *Assume $P \models ER$, we show that $P \models AKR$. Pick any $\pi \in \mathcal{M}(P)$ and $(\pi, i) \in \pi$. Let $\sigma_0 = \sigma(\pi, 0)$, $\tau = trace(\pi, i)$ and $\mathcal{E} = \{\phi_1, \cdots, \phi_k\}$ the set of release whose flag has been set. By Def. 2.6.1, AKR requires only $\mathsf{ESPM}(\mathcal{E})$ to hold at (π, i). By hypothesis and Def. 2.6.2, $\mathcal{R}(\mathcal{M}, \sigma_0, \tau) \subseteq \mathcal{K}(\mathcal{M}, \sigma_0, \tau)$; therefore, for all π' such that $\sigma_0 \approx_{\vec{l}} \sigma(\pi', 0)$ and $\bigwedge_{\phi \in \Phi_{\sigma_0, \tau}} \sigma_0(\phi) = \sigma(\pi', 0)(\phi)$, there exists (π', i') such that $trace(\pi', i') = \tau$. As $\mathcal{D}_{\sigma_0, \tau} \subseteq \mathcal{E}$, it implies $\mathsf{ESPM}(\mathcal{E})$.* □

Figure 2.4 explains the epistemic release wrt. the attacker knowledge. As before, the graphic corresponds to the knowledge about initial secrets that program semantics releases by means of the output trace $\tau = o_1 o_2 o_3$. The black solid line shows how the knowledge can possibly increase in each output point by disclosing information about the secrets. The red dotted line shows the secret information declassified in each epoch by release points r_i. Since the dotted line remains above the solid line, the attacker knowledge is less than what the programmer releases by means of these points. Hence the program will satisfy the security condition.

Example 2.6.3 *Consider a program P (variation of [156]) with secret, $x, y \in \vec{h}$ and in, $l \in \vec{l}$. P allows a local release point r_ϕ with declassification policy $\phi = hash(h) \bmod 2^{64} = in$ i.e. private variable secret can only be leaked comparing the*

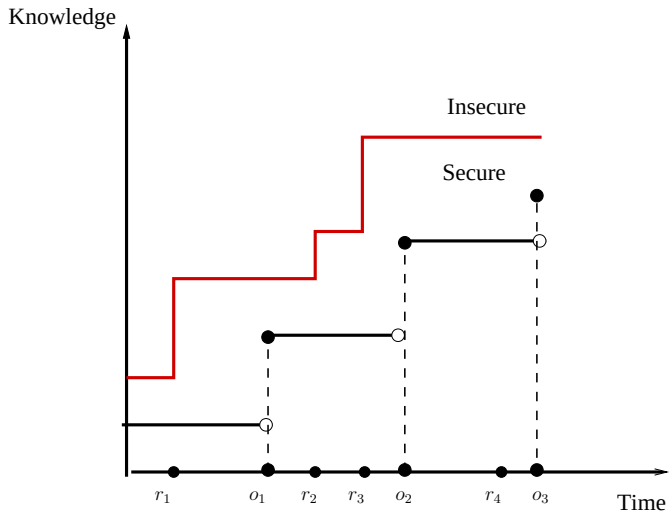


Figure 2.4: Knowledge and Release

least 64 bits of his hashed value to public input variable y.

$$P ::= \left[\begin{array}{l} x := hash(h); y = x \bmod 2^{64}; \\ \texttt{if } y = in \texttt{ then } l := 0 \texttt{ else } l := 1; \\ r_\phi; \texttt{out}(l); \end{array}\right.$$

Applying Def. 2.6.1, one can see that for any fixed initial value of identifiers in, l, for all initial values h having property ϕ the output value is 1 and all initial h having property $\neg\phi$ the output value is 2. However, if we append to P the following lines of code (where $z \in \vec{l}$), it becomes insecure.

$$P' ::= P; z := x \bmod 3; \texttt{out}(z)$$

Indeed, pick h_1, h_2 satisfying ϕ and $hash(h_1) \bmod 3 \neq hash(h_2) \bmod 3$, then it violates the release policy.

2.7 Declassification: When

The last dimension of declassification addressed in this paper is the "when" dimension [202]. Following an approach similar to the one of Chong and Myers [75], a temporal declassification is a pair (ϕ^C, ϕ^D) composed of a declassified property ϕ^D and a time predicate ϕ^C which specifies *when* to declassify ϕ^D. During any execution, as soon as ϕ^C holds, outsiders are allowed to learn ϕ^D now and in the future.

Let Φ be a set of *temporal declassifications*, Φ^C denotes the set of time predicates of Φ ($\Phi^C = \{\phi^C \mid (\phi^C, \phi^D) \in \Phi\}$) and Φ^D denotes the set of declassified properties of Φ. It has to be noted that there are two types of temporal declassifications. If ϕ^C applies to values which are constant during the execution (such as the initial value of a given variable) or are expressed using *init* in our model, (ϕ^C, ϕ^D) describes for which executions an information can be output. A policy stating that a salary can be output only if it is lower than a given constant is an example of such an *inter-execution* temporal declassification. On the other hand, if ϕ^C applies to variables whose value vary during the execution then (ϕ^C, ϕ^D) describes after which event an information can be leaked. An *intra-execution* temporal declassification is for example a policy stating that an information can be provided only after it has been paid for.

Following the standard definitions of NI (Def. 2.4.1) and NID (Def. 2.5.1), Def. 2.7.1 formally defines *noninterference modulo temporal declassifications*. It states that at any point (π_1, i_1) of any execution π_1, for any execution π_2 started with the same initial public values ($\sigma(\pi_1, 0) \approx_{\overline{l}} \sigma(\pi_2, 0)$) and agreeing on declassifications ($\sigma(\pi_1, 0) \approx_{\psi^D} \sigma(\pi_2, 0)$) activated so far ($\exists j : 0 \leq j \leq i_1 \wedge \sigma(\pi_1, j)(\phi^C)$), there should exists a point (π_2, i_2) which has the same trace as (π_1, i_1).

Definition 2.7.1 (NITD)
Let Φ be a a set of temporal declassifications*, i.e. a set of pairs (ϕ_i^C, ϕ_i^D). A program P satisfies* noninterference modulo temporal declassifications Φ *iff:*

$$\forall \pi_1, \pi_2 \in \mathcal{M}(P), \forall (\pi_1, i_1) \in \pi_1 :$$
$$\left(\begin{array}{l} \sigma(\pi_1, 0) \approx_{\overline{l}} \sigma(\pi_2, 0) \quad \wedge \\ \bigwedge\limits_{(\phi^C, \phi^D) \in \Phi} \left\{ \begin{array}{l} \left(\exists j : 0 \leq j \leq i_1 \wedge \sigma(\pi_1, j)\phi^C\right) \\ \Rightarrow \sigma(\pi_1, 0) \approx_{\phi^D} \sigma(\pi_2, 0) \end{array} \right. \end{array} \right)$$
$$\Rightarrow \exists i_2,\ \mathit{trace}(\pi_1, i_1) = \mathit{trace}(\pi_2, i_2)$$

In our framework, this complex predicate can be naturally expressed using once again the ESPM formula. Definition 2.7.2 provides the complete epistemic temporal formula that has to hold in order for a program P to satisfy *absence of knowledge modulo temporal declassifications* Φ.

Definition 2.7.2 (AKTD)
Let Φ be a set of temporal declassifications*. A program P satisfies* absence of knowledge modulo temporal declassifications Φ *iff:*

$$P \models \bigwedge_{\Psi \in \mathcal{P}(\Phi)} \left(\mathsf{ESPM}(\Psi^D)\, W \left(\bigvee_{\phi \in (\Phi \setminus \Psi)^C} \phi \right) \right)$$

For any subset of declassification policies $\Psi \subseteq \Phi$, noninterference modulo declassifications Ψ^D ($\mathsf{ESPM}(\Psi^D)$) has to hold until the condition ϕ^C of an information not

declassified by Ψ^D holds ($\phi^D \notin \Psi^D$). In particular, noninterference ($\mathsf{ESPM}(\emptyset)$ by Prop. 2.5.1) has to hold until the first information is declassified. Generally, if Ψ^C is the set of all declassification conditions which have been triggered so far, noninterference modulo Ψ^D and all superset of Ψ^D has to hold ($\forall \Psi_2^D\colon \mathsf{ESPM}(\Psi^D \cup \Psi_2^D)$). However, by Prop. 2.5.2, noninterference modulo Ψ^D subsumes noninterference modulo any superset of Ψ^D, and is therefore the real policy enforced when the set of conditions triggered so far is Ψ^C.

Proposition 2.7.1 (Equivalence of NITD and AKTD)
For all programs P:

$$P \models \mathit{NITD} \;\; \mathit{iff} \;\; P \models \mathit{AKTD}$$

PROOF. *Let $\Phi_{(\pi,i)} \subseteq \Phi$ be the set of all temporal declassifications (ϕ^C, ϕ^D) which have been triggered at execution point (π, i) ($\exists j : 0 \leq j \leq i \wedge \sigma(\pi, j)\phi^C$).*
(⇒) For all execution points (π_1, i_1) and initial stores σ_2^0 which have the same public values as the initial store of (π_1, i_1) ($\sigma(\pi_1, 0) \approx_{\overline{l}} \sigma_2^0$) and agree on $\Phi^D_{(\pi,i)}$ ($\sigma(\pi_1, 0) \approx_{\Phi^D_{(\pi,i)}} \sigma_2^0$), there exists an execution π_2 started in the initial state σ_2^0 which has the same trace as (π_1, i_1) at some point (π_2, i_2). This follows from Def. 2.7.1, the fact that for all ϕ^C not in $\Phi^C_{(\pi,i)}$ there is no execution point preceding or equal to (π_1, i_1) such that ϕ^C holds, and $\sigma_1 \approx_{\Phi^D_{(\pi,i)}} \sigma_2$ implies $\sigma_1 \approx_{\phi^D} \sigma_2$ for all ϕ^D in $\Phi^D_{(\pi,i)}$.
The above statement corresponds to: $\mathsf{ESPM}(\Phi^D_{(\pi_1,i_1)})$ holds for all point (π_1, i_1) *(Def. 2.5.2). All the rest of the proof follows from it. First showing that for any subset Ψ of Φ and execution point, either* $\mathsf{ESPM}(\Psi)$ *holds (1) or there exists $\phi \in (\Phi \setminus \Psi)$ such that ϕ holds in the current execution point or a preceding one (2). Then, AKTD is proved by contradiction. If AKTD does not hold then there exists a subset Ψ of Φ and an execution point (π, i) such that* $\mathsf{ESPM}(\Psi)$ *does not hold at (π, i), which would contradict (1), and no $\phi \in (\Phi \setminus \Psi)$ is such that ϕ holds in (π, i) or a preceding point, which would contradict (2).*
For any Ψ, Prop. 2.5.2 implies that $\mathsf{ESPM}(\Phi^D_{(\pi_1,i_1)} \cup \Psi)$ *holds at (π_1, i_1). Hence, for any $\Psi \supseteq \Phi^D_{(\pi_1,i_1)}$, (1) holds, and a fortiori (1) or (2). For any $\Psi \in \mathcal{P}(\Phi)$ not superset of $\Phi_{(\pi_1,i_1)}$, there exists $\phi \in \Phi_{(\pi_1,i_1)} \setminus \Psi$ such that ϕ belongs to $\Phi \setminus \Psi$ and holds at (π_1, i_1) or a preceding state. Hence, for any $\Psi \not\supseteq \Phi^D_{(\pi_1,i_1)}$, (2) holds, and a fortiori (1) or (2). Therefore, NITD ⇒ AKTD.*
(⇐) The proof follows in the reverse order the same equivalence relations as above; relying on the fact that for any point (π_1, i_1) $\mathsf{ESPM}(\Phi^D_{(\pi,i)})$ *has to hold.* □

Example 2.7.1 *Let P, whose code is provided below, be a program that outputs a data after payment of its cost.*

```
while paid < cost do {paid := paid + note} ;
if cost > max then out("ok") else out(paid) ;
out(data)
```

Initial value stores $(paid, note, max, cost, data)$ *are of the shape* $(0, \mathtt{n}, \mathtt{m}, \mathtt{c}, \mathtt{d})$ *where* $\mathtt{n}$, $\mathtt{m}$, $\mathtt{c}$ *and* $\mathtt{d}$ *are integers. The intended security policy is that the* initial *values of paid, note and max are public and everything else should be kept secret, except for the cost which can be revealed only if it is not greater than max (note that if cost is not lower than max then the final value of paid must not be revealed either) and data which can be output after payment. In our framework, this policy is formalized by* $paid, note, max \in \vec{l}$ *and* $\Phi = \{(tt, cost > max), (cost \leq max, cost), (paid \geq cost, data)\}$. *The first declassification of* $cost > max$ *may seem unnecessary, however in order to reveal the cost only if* $cost \leq max$ *it is required to declassify* $cost > max$. *Possible traces of P are: "" while still paying, "*ok*" and "*$\mathsf{ok}\ \mathtt{d}$*" if* $\mathtt{c} > \mathtt{m}$, *otherwise "x" and "*$x\ \mathtt{d}$*" where* $x = \mathtt{n} \times \lceil \mathtt{c} \div \mathtt{n} \rceil$. *Obviously, any execution point of P before the first output satisfies noninterference and* $\mathsf{ESPM}(\Psi)$ *for all* Ψ *(Prop. 2.5.1). However, as the time predicate of* $cost > max$ *is tt, AKTD never requires* $\mathsf{ESPM}(\emptyset)$ *to be satisfied. Only* $\mathsf{ESPM}(\{cost > max\})$ *is required to be satisfied at the beginning of the execution if* $\mathtt{c} > \mathtt{m}$, *otherwise* $\mathsf{ESPM}(\{cost > max, cost\})$ *which is equivalent to* $\mathsf{ESPM}(\{cost\})$ *as max contains a public data (any executions started with the same public data and cost have to agree on* $cost > max$*). After the loop, payment has been made and paid* $\geq$ *cost implies that AKTD only requires* $\mathsf{ESPM}(\{cost > max, data\})$ *to be satisfied if* $\mathtt{c} > \mathtt{m}$, *and otherwise* $\mathsf{ESPM}(\{cost > max, cost, data\})$ *which is equivalent to* $\mathsf{ESPM}(\{cost, data\})$. *If* $\mathtt{c} > \mathtt{m}$ *then next traces are "*ok*" and "*$\mathsf{ok}\ \mathtt{d}$*". For any initial value store differing only on cost but such that* $cost > max$, *there exist an execution point whose trace is "*ok*" and another for "*$\mathsf{ok}\ \mathtt{d}$*". For executions where* $\mathtt{c} \leq \mathtt{m}$ *and after the loop, AKTD only requires that executions started with the same initial value store can generate the same trace. Hence, P satisfies AKTD.*

2.8 Conclusion and Future Work

We have pointed out a strong connection between temporal epistemic logic and several security conditions studied in the area of language-based security, including (state-based) noninterference and various flavors of declassification. We claim that temporal epistemic logic appears to be a well suited logical framework to express and study information flow policies. There have been other attempts at building such general frameworks in the past, including McLean's selective interleaving functions [167] and Mantel's modular assembly kit [159]. These approaches are quite different, and focus more on the modular construction of security properties than their extensional properties. Other notable attempts include Banerjee, Naumann and coauthors work on information flow logics (cf. [44] involving various specialized constructs to constrain data flow and dependencies between variables. An interesting feature of the epistemic account of information flow is that indirect flows are handled completely indirectly: it is never necessary to explicitly talk about variables on different executions being in agreement, or depending on each other; information flow is fully captured in terms of the effects of these dependencies on

agents knowledge.

Our approach is not yet general enough to handle general trace-based conditions. This paper considers programs with output events only, whereas most work on trace-based security conditions address traces consisting of both output and input events. There is no problem in principle to extend our approach to programs with both inputs and outputs, e.g. the interactive programs considered by Bohannon et al [57]. Extending the study in this direction to better understand the role and limits of temporal epistemic definability in security modeling is an important line of inquiry for future work.

The reader will have noticed that we actually use only a very small fragment of the logic we set out to study. For instance, we only use the epistemic possibility operator L and never its dual K (epistemic necessity, knowledge), and never use nesting of epistemic connectives. The former is due to our focus on confidentiality rather than integrity properties. Temporal epistemic logic in its standard form may be richer than needed for the application domain; computational or proof-theoretical gains may be made by considering sparser languages. Related to this is the general problem of tractability, and if the temporal epistemic setting can be used to develop techniques for more precise information flow analysis.

Acknowledgements. This work was partially supported by the EU-funded FP7-project HATS (grant № 231620).

Chapter 3

A Logic for Information Flow Analysis of Distributed Programs

Musard Balliu

Abstract

Securing communication in large scale distributed systems is an open problem. When multiple principals exchange sensitive information over a network, security and privacy issues arise immediately. For instance, in an online auction system we may want to ensure that no bidder knows the bids of any other bidder before the auction is closed. Such systems are typically interactive/reactive and communication is mostly asynchronous, lossy or unordered. Language-based security provides language mechanisms for enforcing end-to-end security. However, with few exceptions, previous research has mainly focused on relational or synchronous models, which are generally not suitable for distributed systems.

This paper proposes a general knowledge-based account of possibilistic security from a language perspective and shows how existing trace-based conditions fit in. A syntactic characterization of these conditions, given by an epistemic temporal logic, shows that existing model checking tools can be used to enforce security.

3.1 Introduction

The emergence of ubiquitous computing paradigm makes software security more and more a real concern. Web browsers, smartphones, clouds are only few examples where untrusted and partially trusted code is regularly executed alongside applications processing personal sensitive data. In addition, current trends in computing such as code mobility and platform independence make the situation even

worse. Attackers can then exploit vulnerabilities and deduce information about sensitive data by observing the behavior of malicious, or simply buggy, programs.

Information flow security policies [122], if successfully enforced or verified, prevent different types of confidentiality and integrity attacks. Language-based security provides end-to-end guarantees by means of programming language techniques. However, most work on language-based security models of information flow assumes synchronous or relational communication [100, 197, 182]. Although these models are important in many settings, they are not obviously well suited for distributed programs where communication is interactive/reactive, nondeterministic and mostly asynchronous, lossy or unordered. The result is that programs that are considered insecure in one model may be secure in another, and vice versa.

Moreover, information flow policies are hard to verify in practice. The majority of static analyses for information flow security use standard methods such as security type systems [218, 197]. These analyses are efficient and attempt to ensure a strict separation, up to declassification and endorsement, between the sensitive part of the computation and the observable part of the computation. Obviously, if both parts are separated, it is impossible to learn anything about the sensitive data by observing the public data. Despite their efficiency, these methods lack the precision needed to handle programs where public and sensitive information are securely interwoven. Few works [38, 37, 128], at least in the setting of software security, attempt to deduce what is learned by observing the public effects of the computation, and then verify that the acquired knowledge does not break a given information flow policy toward sensitive data.

Motivating Example An online auction is a distributed system consisting of an auctioneer A and several bidders B_i competing for items I_k. Such systems are complex and usually involve both message passing and shared memory. For example, the auctioneer may receive messages from bidders who want to participate in the auction and associate a dedicated thread to each request. Then, depending on the auction protocol, each thread may read and write to a private shared array containing bids for all bidders and items. Several information flow policies may be worth enforcing in this scenario[1].

P_1 : The authentication code (pwd) of bidder B_i is always (G) secret wrt. any bidder B_j. In logic: $G\neg K_{B_j}(pwd_{B_i} = v)$.

P_2 : The sequence of bids of bidder B_i remains secret wrt. all bidders B_j until (W) the auction is closed. In logic: $L_{B_j}(secArray = v)\ W\ aClosed$.

P_3 : Only the first 3 bids of bidder B_i are considered secret wrt. any bidder B_j until the auction is closed. In logic: $L_{B_j}(\phi(b_1^i, b_2^i, b_3^i))\ W\ aClosed$.

P_4 : Any bid of bidder B_3 remains secret wrt. a colluding attack of B_1 and B_2. In logic: $GL_{B_1,B_2}(b^3 = v)$.

[1]The reader can already get the flavor of the logic used for security specifications.

P_5 : The system may nondeterministically select a subset of bids from the private array, compute the maximum and promote an item I^* to the winner B^*. The output of this process may be considered secret wrt. any bidder $B_j \neq B^*$. In logic: $G\neg K_{B_j}(out(B^*, I^*))$.

As illustrated above, several issues should be handled to enforce the security policies of such systems. First, they are inherently nondeterministic, hence possibilistic notions of information flow security are needed. Second, distributed programs are usually interactive/reactive, which requires protection of sequences of (input or output) events as opposed to classical relational models where the input is read in the beginning of execution. Third, security policies are usually dynamic and involve controlled release of secret information. Finally, in distributed settings attackers may collude and share their observations to disclose secret information.

In this paper we model distributed systems in a trace-based setting where an execution trace is a sequence of events on channels. Security properties are expressed in terms of knowledge-based (epistemic) conditions over system traces. The security model brings out what events $\mathcal{O}$ on channels an observer can see and what observations on events $\mathcal{P}$ should be protected. Then the system is secure if the knowledge about events in $\mathcal{P}$ of an observer who makes observations in $\mathcal{O}$, at any point in the execution trace, is in accordance with the security policy at that point. Namely, the observer is unable to learn more information than what is allowed at a given point while moving to a successive point of the same trace and possibly making a new observation. This model fits well with current knowledge-based approaches to information flow security [28, 38, 24], and, inspired by work of Guttman and Nadel [127], by being explicit about the information that needs to be protected, it allows a very general treatment of secret information, both as high level input and output events, and as relationships between events, say ordering, multiplicity, and interleaving. We show that several possibilistic conditions such as Separability, Generalized Noninterference, Nondeducibility, Nondeducibility on Outputs and Nondeducibility on Strategies are accurately reflected in the epistemic setting.

Then we turn to the verification problem and present a linear time epistemic logic, with past time operators, which allows us to syntactically characterize security properties. The logic can be used as specification language for expressing possibilistic information flow policies. This enables modeling of the intricate and precise policies described in the motivating example and, at the same time, ensures separation between the actual code and the policy. Recent advances software model checking and automated theorem proving show that verification of temporal epistemic properties for distributed systems is feasible [141]. Our tool, ENCoVer [39], an extension of Java Pathfinder [184], can verify information flow policies for interactive sequential programs. However, scalability and complexity of verification are issues that we postpone to future work.

3.2 Security Model

Program Model A model $\mathcal{M}$ is a set of finite or infinite traces induced by the program semantics. A trace τ is a sequence of actions relevant to the analysis. For instance, it can be messages sent over channels, read/write operations to shared memory, logical time ticks and so on. We write $|\tau|$ for the length (number of actions) of the trace τ. Whenever $|\tau| = \infty$, the trace has infinite length. A *point* is a pair (τ, i), where τ is a trace and $0 \leq i \leq |\tau|$. The function *trace* maps trace points to the prefix of the trace up to that point, namely $\mathit{trace}(\tau, i)$ denotes the sequence of actions α_j, where $0 \leq j < i$. In our setting, the actions belong to a set $\mathit{Act} = \{\mathbf{out}(c, v), \mathbf{in}(c, v) \mid v \in \mathit{Val}, c \in \mathit{Chan}\} \cup \{\epsilon\}$, where Val is the domain of values and $\mathbf{in}(c, v)$ (resp. $\mathbf{out}(c, v)$) denotes the input (output) of value v on channel $c \in \mathit{Chan}$. The silent action is ϵ. We write $\tau_1 \bullet \tau_2$ for concatenation of two traces and $\tau \bullet \alpha$ for concatenation of trace τ with action α. The empty trace is ε and trace interleaving $\ltimes(\tau_1, \tau_2)$ is a set of traces coinductively defined as expected. The set inclusion is denoted as $\preceq$ and $\tau(i)$ is the i-th action of trace τ. The projection of a trace τ on a set of actions $\mathcal{A} \subseteq \mathit{Act}$ is defined as the subsequence of actions from $\mathcal{A}$ and denoted as $\tau_{\downarrow\mathcal{A}}$. Models can be enriched with structure by defining particular relations. In particular, given a poset $(S, \sqsubseteq)$, an upper closure operator (for short *uco*) is a function $\rho : S \to S$ such that $(a)\forall s \in S.\ s \sqsubseteq \rho(s)$, $(b)\forall s_1, s_2 \in S.\ s_1 \sqsubseteq s_2 \Rightarrow \rho(s_1) \sqsubseteq \rho(s_2)$, $(c)\forall s \in S.\ \rho(s) = \rho(\rho(s))$.

Security Policy We are mainly concerned with protecting confidentiality of actions on channels, hence we assume a set of security levels $\mathcal{L}$ for confidentiality and a relation $\sqsubseteq$ over $\mathcal{L}$. Moreover, we consider two observers (potentially sets of agents), one of security level `H` and the other of security level `L`, which interact with the system by providing inputs and receiving outputs on channels of the same security level. Each agent has different clearance represented by a poset $(\mathcal{L}, \sqsubseteq)$ of two elements $\mathcal{L} = \{\mathtt{H}, \mathtt{L}\}$ with $\mathtt{L} \sqsubseteq \mathtt{H}$. A partial function $\mathcal{S} : \mathit{Chan} \rightharpoonup \mathcal{L}$, mapping channels to security levels, determines the set of channels accessible to each observer. Then the security policy is defined as a pair $\mathcal{P}ol = (\mathcal{O}, \mathcal{P})$ where $\mathcal{O}$ is the set of channels that an observer can control and $\mathcal{P}$ is the set of channels to be protected. Usually we define $\mathcal{O} = \{c \in \mathit{Chan} \mid \mathcal{S}(c) = \mathtt{L}\}$ and $\mathcal{P} = \{c \in \mathit{Chan} \mid \mathcal{S}(c) = \mathtt{H}\}$. The fact that $\mathcal{S}$ is partially defined allows us to model channels which are invisible to the observer, yet not subject to protection.

Security Condition The security condition determines when a model $\mathcal{M}$ is secure with respect to a security policy $(\mathcal{O}, \mathcal{P})$. Here we define security in terms of the knowledge of an observer who knows the system specification[2] and interacts through channels in $\mathcal{O}$. The security condition prevents the observer from learning information about (properties of) interactions through channels in $\mathcal{P}$. First we

[2] In a language-based security setting the attacker is usually assumed to have complete knowledge of the program code.

define the observer knowledge at point (τ, i) as

$$\mathcal{K}(\tau, i, \mathcal{O}) = \{\tau' \mid \tau' \in \mathcal{M} \wedge (\tau', i') =_{\downarrow\mathcal{O}} (\tau, i) \wedge |i - i'| \leq t\}$$

where t is a *synchrony* parameter of the observer, $0 \leq i' \leq |\tau|$ and $(\tau', i') =_{\downarrow\mathcal{O}} (\tau, i)$ if the projection of (τ, i) and (τ', i') on actions in $\mathcal{O}$ is the same, namely, $(\tau', i')_{\downarrow\mathcal{O}} = (\tau, i)_{\downarrow\mathcal{O}}$. Intuitively, $\mathcal{K}(\tau, i, \mathcal{O})$ represents the set of traces that the observer considers possible based on its observations up to point (τ, i) and having synchrony parameter t. In particular, in a synchronous system $t = 0$, i.e. the observer knows the exact logical time. An asynchronous system can similarly be modeled by $t = \infty$. Models of semi-synchronous systems, where the observer knows the time approximately, can also be expressed. Then we define projection of trace τ on a set of channels $\mathcal{C}$, where $e(c, v)$ denotes an event on channel c.

$$\tau_{\downarrow\mathcal{C}} ::= \begin{cases} \varepsilon & \text{if } \tau = \varepsilon \\ e(c, v) :: \tau'_{\downarrow\mathcal{C}} & \text{if } c \in \mathcal{C} \text{ and } \tau = e(c, v) \bullet \tau' \\ \tau'_{\downarrow\mathcal{C}} & \text{if } c \notin \mathcal{C} \text{ and } \tau = e(c, v) \bullet \tau' \end{cases}$$

Notice that we leave the meaning of the :: operator undefined. By default we interpret :: as concatenation, however it need not be, as we will see when discussing different security policies.

At this point we have all ingredients to present the knowledge-based security condition. The intuition is simple: given a model $\mathcal{M}$ and a security policy $\mathcal{P}ol = (\mathcal{O}, \mathcal{P})$, the condition ensures that for each point $i + 1$ of trace τ, the observer's knowledge about actions in $\mathcal{P}$ is not greater than its knowledge at the previous point i.

Definition 3.2.1 (Knowledge-based Security) *Let $\mathcal{M}$ be a model and $\mathcal{P}ol = (\mathcal{O}, \mathcal{P})$ a security policy. Then $\mathcal{M}$ is* secure *wrt. $\mathcal{P}ol$ if for all $\tau \in \mathcal{M}$, $0 \leq i < |\tau|$*

$$\mathcal{K}(\tau, i, \mathcal{O})_{\downarrow\mathcal{P}} \preceq \mathcal{K}(\tau, i + 1, \mathcal{O})_{\downarrow\mathcal{P}}$$

We illustrate the main idea behind the security condition with an example.

Example 3.2.1 *Consider a program $P ::= \mathbf{in}(c_1, x); \mathbf{out}(c_2, x)$ which receives a boolean value on input channel c_1 and sends it on output channel c_2. The model $\mathcal{M}_P$ of P consists of two traces $\tau_1 = \mathbf{in}(c_1, true) \bullet \mathbf{out}(c_2, true)$ and $\tau_2 = \mathbf{in}(c_1, false) \bullet \mathbf{out}(c_2, false)$. The goal is to check whether $\mathcal{M}_P$ satisfies the policy $\mathcal{P}ol = (\mathcal{O}, \mathcal{P})$, where $\mathcal{O} = \{c_2\}$ and $\mathcal{P} = \{c_1\}$. Namely, we check if agent $\mathtt{L}$, the attacker, who knows $\mathcal{M}_P$ and observes values on channel c_2, can deduce information about activity of agent $\mathtt{H}$ on channel c_1. We identify agent $\mathtt{L}$ with $\mathcal{O}$ and agent $\mathtt{H}$ with $\mathcal{P}$. Then, applying Def. 3.2.2, we obtain ($k \in \{1, 2\}$):*

- $\mathcal{K}(\tau_k, 0, \mathtt{L})_{\downarrow\mathtt{H}} = \mathcal{K}(\tau_k, 1, \mathtt{L})_{\downarrow\mathtt{H}} = \{\mathbf{in}(c_1, true), \mathbf{in}(c_1, false)\}$
- $\mathcal{K}(\tau_1, 2, \mathtt{L})_{\downarrow\mathtt{H}} = \{\mathbf{in}(c_1, true)\}$ *and* $\mathcal{K}(\tau_2, 2, \mathtt{L})_{\downarrow\mathtt{H}} = \{\mathbf{in}(c_1, false)\}$

The program is insecure since $\mathcal{K}(\tau_1, 1, \mathtt{L})_{\downarrow \mathtt{H}} \not\preceq \mathcal{K}(\tau_1, 2, \mathtt{L})_{\downarrow \mathtt{H}}$. *Namely, when the attacker observes* $\mathbf{out}(c_2, \mathit{true})$, *he refines his knowledge about secret actions from* $\{\mathbf{in}(c_1, \mathit{true}), \mathbf{in}(c_1, \mathit{false})\}$ *to* $\{\mathbf{in}(c_1, \mathit{true})\}$ *and deduces that value* true *was input on channel* c_1 *by agent* H.

The security condition in Def. 3.2.1 can be relaxed to deal with different forms of dynamic policies [202, 24]. A release (or declassification) policy $\mathcal{R}(\tau, i, \mathcal{P})$ at point (τ, i) is a property of $\mathcal{P}$, i.e., subset of $\mathcal{M}_{\downarrow \mathcal{P}}$, representing the knowledge that the observer is allowed to learn at that point. Consequently, a program is secure if the attacker's knowledge and the released knowledge at point (τ, i) is not greater than the attacker's knowledge at point $(\tau, i+1)$.

Definition 3.2.2 (Security wrt. Release) *Let* $\mathcal{M}$ *be a model with security policy* $\mathcal{P}ol = (\mathcal{O}, \mathcal{P})$ *and release policy* $\mathcal{R}(\tau, i, \mathcal{P})$. *Then* $\mathcal{M}$ *is* secure *wrt.* $\mathcal{P}ol$ *and* $\mathcal{R}(\tau, i, \mathcal{P})$ *if for all* $\tau \in \mathcal{M}$, $0 \leq i < |\tau|$

$$\mathcal{K}(\tau, i, \mathcal{O})_{\downarrow \mathcal{P}} \cap \mathcal{R}(\tau, i, \mathcal{P}) \preceq \mathcal{K}(\tau, i+1, \mathcal{O})_{\downarrow \mathcal{P}}$$

If no action from $\mathcal{P}$ is released, $\mathcal{R}(\tau, i, \mathcal{P}) = \mathcal{M}_{\downarrow \mathcal{P}}$ and Def. 3.2.1 and Def. 3.2.2 coincide. Reconsider Ex. 3.2.1 with release policy $\mathcal{R}(\tau_1, 1, \mathtt{H}) = \{\mathbf{in}(c_1, \mathit{true})\}$ and $\mathcal{R}(\tau_2, 1, \mathtt{H}) = \{\mathbf{in}(c_1, \mathit{false})\}$. Then P is secure as $\mathcal{K}(\tau_1, 1, \mathtt{L})_{\downarrow \mathtt{H}} \cap \{\mathbf{in}(c_1, \mathit{true})\} \preceq \mathcal{K}(\tau_1, 2, \mathtt{L})_{\downarrow \mathtt{H}}$ and $\mathcal{K}(\tau_2, 1, \mathtt{L})_{\downarrow \mathtt{H}} \cap \{\mathbf{in}(c_1, \mathit{false})\} \preceq \mathcal{K}(\tau_2, 2, \mathtt{L})_{\downarrow \mathtt{H}}$.

In a distributed setting, different agents may form coalitions and share observations in order to disclose secret information about other agents. The following definition gives a security condition in presence of colluding attacks.

Definition 3.2.3 (Security wrt. Collusion) *Let* $\mathcal{M}$ *be a model and two agents* a_1, a_2 *observing, resp.,* $\mathcal{O}_1, \mathcal{O}_2$. *Then* $\mathcal{M}$ *secure wrt. a* colluding attack *on* $\mathcal{P}$ *if* $\mathcal{M}$ *is secure wrt. policy* $(\mathcal{O}_1 \cup \mathcal{O}_2, \mathcal{P})$.

Example 3.2.2 *Let* P *be a program with* $c_1 \in \mathcal{P}$, $c_2 \in \mathcal{O}_1$ *and* $c_3 \in \mathcal{O}_2$. P *is secure wrt. policies* $\mathcal{P}ol_1 = (\mathcal{O}_1, \mathcal{P})$ *and* $\mathcal{P}ol_2 = (\mathcal{O}_2, \mathcal{P})$, *but insecure wrt. a colluding attack, i.e., the policy* $\mathcal{P}ol = (\mathcal{O}_1 \cup \mathcal{O}_2, \mathcal{P})$.

$$P ::= \left[\begin{array}{l} \mathbf{in}(c_1, x) \\ \mathtt{if}\ x\ \mathtt{then}\ \mathbf{out}(c_2, 0) || \mathbf{out}(c_3, 1) \\ \mathtt{else}\ \mathbf{out}(c_2, 0); \mathbf{out}(c_3, 1) \end{array}\right.$$

To see this, consider the program model $\mathcal{M}_P = \{\mathbf{in}(c_1, 1) \bullet \mathbf{out}(c_2, 0) \bullet \mathbf{out}(c_3, 1),$ $\mathbf{in}(c_1, 1) \bullet \mathbf{out}(c_3, 1) \bullet \mathbf{out}(c_2, 0),$ $\mathbf{in}(c_1, 0) \bullet \mathbf{out}(c_2, 0) \bullet \mathbf{out}(c_3, 1)\}$ *where the secret on* c_1 *is binary and* $||$ *denotes the nondeterministic choice. If an agent merely observes the value received on his channel, there is nothing he can tell about the secret bit on* c_1. *However, if they collude, the observation of low sequence* $\mathbf{out}(c_3, 1) \bullet \mathbf{out}(c_2, 0)$ *reveals that the secret bit was* 1.

Trace-based Conditions We next introduce several possibilistic information flow conditions from the literature and briefly discuss the flavor of each. The reason is two-fold; first to identify which aspects of security they enforce and, second, to show how these aspects can be captured by the knowledge-based conditions. We denote projection of trace τ on a set $\mathcal{A}$ as $\tau_{\mathcal{A}}$ (instead of $\tau_{\downarrow\mathcal{A}}$) to distinguish from the knowledge-based condition.

Separability was first introduced by McLean [164]. The goal is to ensure a logical separation between secret and public computations in both directions.

Definition 3.2.4 (Sep) *A model* $\mathcal{M}$ *satisfies* separability *if*

$$\forall \tau, \tau' \in \mathcal{M}, \forall \tau^* \in \ltimes(\tau_{\mathtt{L}}, \tau'_{\mathtt{H}}), \tau^* \in \mathcal{M}$$

A version of separability for synchronous systems has been proposed in [128].

Definition 3.2.5 (SSep) *A model* $\mathcal{M}$ *satisfies* synchronous separability *if*

$$\forall \tau, \tau' \in \mathcal{M},\ \exists \tau^* \in \mathcal{M}.\ \tau^*_{\mathtt{L}} = \tau_{\mathtt{L}} \text{ and } \tau^*_{\mathtt{H}} = \tau'_{\mathtt{H}}$$

Generalized noninterference is a relaxation of separability and it ensures that low computation is independent of the sequence of high inputs `HI` [164].

Definition 3.2.6 (GNI) *Model* $\mathcal{M}$ *satisfies* generalized noninterference *if*

$$\forall \tau, \tau' \in \mathcal{M}, \forall \tau^* \in \ltimes(\tau_{\mathtt{L}}, \tau'_{\mathtt{HI}}),\ \exists \tau'' \in \mathcal{M},\ \tau^* = \tau''_{\mathtt{L}\cup\mathtt{HI}}$$

Moreover, GNI prevents the low user from deducing information about both occurrences and non occurrences of high inputs. Nondeducibility, introduced by Sutherland [211], considers the system as a set of possible worlds W and defines security in terms of (information) functions f and g, such that for all $w_1, w_2 \in W$, there exists $w_3 \in W$ and $f(w_1) = f(w_3)$ and $g(w_2) = g(w_3)$. If one interprets f as computing the sequence of high input events and g as computing the sequence of low events, then nondeducibility can be defined as follows.

Definition 3.2.7 (ND) *A model* $\mathcal{M}$ *satisfies* nondeducibility *if*

$$\forall \tau, \tau' \in \mathcal{M},\ \exists \tau^* \in \mathcal{M}.\ \tau^*_{\mathtt{HI}} = \tau_{\mathtt{HI}} \text{ and } \tau^*_{\mathtt{L}} = \tau'_{\mathtt{L}}$$

One drawback of GNI and ND is that they are not adequate for systems that need to protect high output events or generate secrets internally. To solve this issue Guttman and Nadel introduced nondeducibility on outputs which prevents deductions of high events [127] and allows information flowing from low user inputs, here `LI`, to high outputs.

Definition 3.2.8 (NDO) *A model* $\mathcal{M}$ *satisfies* nondeducibility on outputs *if*

$$\forall \tau, \tau' \in \mathcal{M},\ \tau_{\mathtt{LI}} = \tau'_{\mathtt{LI}},\ \exists \tau^* \in \mathcal{M}.\ \tau^*_{\mathtt{H}\cup\mathtt{LI}} = \tau_{\mathtt{H}\cup\mathtt{LI}} \text{ and } \tau^*_{\mathtt{L}} = \tau'_{\mathtt{L}}$$

On the other hand, ND and GNI are too weak to ensure security for systems that exploit internal nondeterminism to transmit secrets through strategies implemented by high users [224]. A strategy is a function from sequences of high inputs and high outputs to values in a domain. A high user can use a strategy to compute the next input value on a high channel, as a function of the history of high values, and transmit information to a low user.

Definition 3.2.9 (NDS) *Let $\mathcal{M}$ be a model and $s_1, s_2 : \mathcal{M} \to Val$ two high strategies. Then $\mathcal{M}$ satisfies* nondeducibility on strategies *if*

$$\forall \tau, \tau' \in \mathcal{M},\ s_1(\tau_{\mathtt{L}}) = s_2(\tau'_{\mathtt{L}}) \Rightarrow \tau_{\mathtt{L}} = \tau'_{\mathtt{L}}$$

Most of the trace-based conditions assume either synchronous or asynchronous models. However, in language-based security, the knowledge of program code can give partial information about the order of events on high and low channels,and yet the program can be considered secure. We illustrate this fact with an example.

Example 3.2.3 *Consider program P where $\{c_1, c_2\} \in \mathcal{P}$ and $\{c_3\} \in \mathcal{O}$.*

$$\mathbf{in}(c_1, secret); \mathbf{out}(c_2, secret); \mathbf{out}(c_3, \textit{"Done"})$$

The program receives a secret input from a high agent, writes to a file of the same agent and notifies the low agent that the operation is completed. In an asynchronous model, the secret is first received on c_1 and it is sent on c_2 or c_3 in any order. Hence $\mathcal{M}$ consists of the following traces:

$$\mathbf{in}(c_1, v_i) \bullet \mathbf{out}(c_2, v_i) \bullet \mathbf{out}(c_3, \textit{"Done"}),\ \mathbf{in}(c_1, v_i) \bullet \mathbf{out}(c_3, \textit{"Done"}) \bullet \mathbf{out}(c_2, v_i)$$

It can be easily checked that $\mathcal{M}$ does not satisfy Sep and GNI, since the system is not closed under interleavings between H *and* L *events. On the other hand, it seems reasonable to accept P as secure. The knowledge-based condition in Def. 3.2.1 accepts $\mathcal{M}$ wrt. policy $(\mathcal{O}, \mathcal{P})$ as do ND and NDO.*

This example raises the question of what security policy a condition is enforcing and how these conditions can be interpreted in a unified framework.

3.3 Policies via Examples

We now introduce, by means of examples, different security policies using the epistemic security conditions. The set of channels in $\mathcal{P}$ and the set of channels in $\mathcal{O}$ are, resp., identified with H and L. Moreover, we redefine the semantics of the :: operator to handle different policies. We always define :: as concatenation when projecting on channels in $\mathcal{O}$. This reflects the assumption of perfect recall attacker with unbounded memory. Furthermore, given the set of high channels in $\mathcal{P}$, we write Set(H) or just H to define :: as set union and Mul(H) to define :: as multiset union. Finally, Seq(H) defines :: as concatenation, while Seq(H $\perp$) defines :: as

concatenation and replaces events in $\mathtt{L}$ with the special symbol $\perp$. For example, $(\mathtt{L}, \mathit{Mul}(\mathtt{H}))$ defines a policy which protects the multiplicity of high actions wrt. an attacker that observes actions in $\mathcal{O}$. Likewise, the policy $(\mathtt{L}, \mathit{Seq}(\mathtt{H}\ \perp))$ prevents the attacker from deducing information about interleavings of high actions in $\mathcal{P}$ with low actions in $\mathcal{O}$. Whenever the action type is unimportant, we write $l_1, l_2, \cdots$ for actions in $\mathtt{L}$ and $h_1, h_2, \cdots$ for actions in $\mathtt{H}$. In the examples, traces are numbered as $\tau_1, \tau_2, \cdots$ following the order they appear in $\mathcal{M}$.

The first point we want to make is what happens in relational models of information flow where inputs are read in the beginning of program execution. All direct and implicit flows from high channels to low channels are captured by the security policy $(\mathtt{L}, \mathtt{H})$.

Example 3.3.1 *Let a model $\mathcal{M}$ consist of two traces $\mathcal{M} = \{h_1 \bullet h_2 \bullet l_1,\ \ h_1 \bullet h_3 \bullet l_2\}$. Is $\mathcal{M}$ secure wrt. policy $\mathcal{P}ol = (\mathtt{L}, \mathtt{H})$? Intuitively, the answer should be negative as an attacker can associate h_2 with observation l_1 and h_3 with observation l_2. Applying Def. 3.2.1, with $0 \leq i \leq 2$, we obtain*

- $\mathcal{K}(\tau_1, i, \mathtt{L})_{\downarrow \mathtt{H}} = \mathcal{K}(\tau_2, i, \mathtt{L})_{\downarrow \mathtt{H}} = \{h_1, h_2, h_3\}$
- $\mathcal{K}(\tau_1, 3, \mathtt{L})_{\downarrow \mathtt{H}} = \{h_1 \bullet h_2 \bullet l_1\}_{\downarrow \mathtt{H}} = \{h_1, h_2\}$
- $\mathcal{K}(\tau_2, 3, \mathtt{L})_{\downarrow \mathtt{H}} = \{h_1 \bullet h_3 \bullet l_2\}_{\downarrow \mathtt{H}} = \{h_1, h_3\}$

The program is insecure as $\mathcal{K}(\tau_1, 2, \mathtt{L})_{\downarrow \mathtt{H}} \not\preceq \mathcal{K}(\tau_1, 3, \mathtt{L})_{\downarrow \mathtt{H}}$, i.e., $\{h_1, h_2, h_3\} \not\preceq \{h_1, h_2\}$. Using the same policy, another model $\mathcal{M}' = \{h_1 \bullet l_1, h_2 \bullet l_1\}$ is secure.

Example 3.3.2 *Consider now $\mathcal{M} = \{h_1 \bullet h_1 \bullet l_1, h_1 \bullet l_2\}$ wrt. security policy $\mathcal{P}ol = (\mathtt{L}, \mathtt{H})$. Applying Def. 3.2.1, the model is secure. However, there may be cases where $\mathcal{M}$ is considered insecure. For instance, a low user $\mathtt{L}$ may only be interested in knowing when exactly the system is logging his actions, so that an attack can be performed stealthy. To capture these cases it is enough to consider a policy $\mathcal{P}ol_1 = (\mathtt{L}, \mathit{Mul}(\mathtt{H}))$ or $\mathcal{P}ol_2 = (\mathtt{L}, \mathit{Seq}(\mathtt{H}))$. Let $i \in \{0, 1\}$, then*

- $\mathcal{K}(\tau_1, i, \mathtt{L})_{\downarrow \mathit{Seq}(\mathtt{H})} = \mathcal{K}(\tau_2, i, \mathtt{L})_{\downarrow \mathit{Seq}(\mathtt{H})} = \{h_1 \bullet h_1, h_1\}$
- $\mathcal{K}(\tau_1, 2, \mathtt{L})_{\downarrow \mathit{Seq}(\mathtt{H})} = \{h_1 \bullet h_1, h_1\}$
- $\mathcal{K}(\tau_2, 2, \mathtt{L})_{\downarrow \mathit{Seq}(\mathtt{H})} = \{h_1 \bullet l_2\}_{\downarrow \mathit{Seq}(\mathtt{H})} = \{h_1\}$
- $\mathcal{K}(\tau_1, 3, \mathtt{L})_{\downarrow \mathit{Seq}(\mathtt{H})} = \{h_1 \bullet h_1\}$

Clearly, $\mathcal{K}(\tau_2, 1, \mathtt{L})_{\downarrow \mathit{Seq}(\mathtt{H})} \not\preceq \mathcal{K}(\tau_2, 2, \mathtt{L})_{\downarrow \mathit{Seq}(\mathtt{H})}$ as $\{h_1 \bullet h_1, h_1\} \not\preceq \{h_1\}$.

It is worth noting that protecting $\mathtt{H}$ and $\mathit{Mul}(\mathtt{H})$ is of little interest for reactive systems that may receive inputs on the high channels in different order. The following program is considered secure wrt. both $(\mathtt{L}, \mathtt{H})$ and $(\mathtt{L}, \mathit{Mul}(\mathtt{H}))$.

$$P ::= \left[\begin{array}{l} b := \mathit{true} \,||\, \mathit{false} \\ \mathtt{if}\ b\ \mathtt{then}\ \mathbf{in}(c', x); \mathbf{out}(c, 1); \mathbf{in}(c'', y); \mathbf{out}(c, 3) \\ \mathtt{else}\ \mathbf{in}(c'', y); \mathbf{out}(c, 2); \mathbf{in}(c', x); \mathbf{out}(c, 4) \end{array}\right.$$

Indeed, if c' and c'' are high channels and c is a low channel, then $\mathcal{M}_P = \{h_1 \bullet l_1 \bullet h_2 \bullet l_3, h_2 \bullet l_2 \bullet h_1 \bullet l_4\}$ is secure wrt. the above policies. However, when an attacker observes l_1 he knows h_1, i.e. that the first high input was received on channel c', and similarly, when an attacker observes l_3 he knows h_2, i.e. that first high input was was received on channel c''. Hence, the policy $(\mathtt{L}, \mathit{Seq}(\mathtt{H}))$ is needed to rule out this program.

Example 3.3.3 *Let $\mathcal{M} = \{h_1 \bullet h_2 \bullet l_1, h_1 \bullet l_2 \bullet h_2\}$ be a program model. $\mathcal{M}$ is secure wrt. policies in previous examples since the sequence of high actions is the same for both traces. However, an attacker observing l_1 knows that $h_1 \bullet h_2$ has occurred, while this is not ensured if he observes l_2. Similar security issues may arise in scenarios discussed in [127]. To capture such flows, we consider a stronger policy $(\mathtt{L}, \mathit{Seq}(\mathtt{H}\ \bot))$ which protects the interleavings between high actions and occurrences of low actions. Let $i \in \{0, 1\}$, then $\mathcal{M}$ is insecure*

- $\mathcal{K}(\tau_1, i, \mathtt{L})_{\downarrow \mathit{Seq}(\mathtt{H}\bot)} = \mathcal{K}(\tau_2, i, \mathtt{L})_{\downarrow \mathit{Seq}(\mathtt{H}\bot)} = \mathcal{K}(\tau_1, 2, \mathtt{L})_{\downarrow \mathit{Seq}(\mathtt{H}\bot)}$ $= \{h_1 \bullet h_2 \bullet \bot, h_1 \bullet \bot \bullet h_2\}$
- $\mathcal{K}(\tau_1, 3, \mathtt{L})_{\downarrow \mathit{Seq}(\mathtt{H}\bot)} = \{h_1 \bullet h_2 \bullet l_1\}_{\downarrow \mathit{Seq}(\mathtt{H}\bot)} = \{h_1 \bullet h_2 \bullet \bot\}$
- $\mathcal{K}(\tau_2, 2, \mathtt{L})_{\downarrow \mathit{Seq}(\mathtt{H}\bot)} = \mathcal{K}(\tau_2, 3, \mathtt{L})_{\downarrow \mathit{Seq}(\mathtt{H}\bot)} = \{h_1 \bullet \bot \bullet h_2\}$

$\mathcal{K}(\tau_2, 1, \mathtt{L})_{\downarrow \mathit{Seq}(\mathtt{H}\bot)} \not\preceq \mathcal{K}(\tau_2, 2, \mathtt{L})_{\downarrow \mathit{Seq}(\mathtt{H}\bot)}$ *i.e.* $\{h_1 \bullet h_2 \bullet \bot, h_1 \bullet \bot \bullet h_2\} \not\preceq \{h_1 \bullet \bot \bullet h_2\}$

Dynamic Policies and Declassification The security condition in Def. 3.2.1 is too strong to be useful in scenarios where high actions are released intentionally. This is typically the case of dynamic policies where information can be downgraded or upgraded with time.

Example 3.3.4 *Let $\mathcal{M} = \{h_1 \bullet l_1 \bullet h_2 \bullet l_2,\ h_1 \bullet l_3 \bullet h_3 \bullet l_4\}$ be a model with two traces. $\mathcal{M}$ is insecure as the attacker can distinguish h_2 from h_3 after observing, respectively, l_2 and l_4. This is captured by the policy $(\mathtt{L}, \mathtt{H})$.*

- $\mathcal{K}(\tau_i, 0, \mathtt{L})_{\downarrow \mathtt{H}} = \mathcal{K}(\tau_i, 1, \mathtt{L})_{\downarrow \mathtt{H}} = \{h_1, h_2, h_3\}$
- $\mathcal{K}(\tau_1, 2, \mathtt{L})_{\downarrow \mathtt{H}} = \mathcal{K}(\tau_1, 3, \mathtt{L})_{\downarrow \mathtt{H}} = \mathcal{K}(\tau_1, 4, \mathtt{L})_{\downarrow \mathtt{H}} = \{h_1, h_2\}$
- $\mathcal{K}(\tau_2, 2, \mathtt{L})_{\downarrow \mathtt{H}} = \mathcal{K}(\tau_2, 3, \mathtt{L})_{\downarrow \mathtt{H}} = \mathcal{K}(\tau_2, 4, \mathtt{L})_{\downarrow \mathtt{H}} = \{h_1, h_3\}$

It is clear that $\mathcal{K}(\tau_1, 1, \mathtt{L})_{\downarrow \mathtt{H}} \not\preceq \mathcal{K}(\tau_1, 2, \mathtt{L})_{\downarrow \mathtt{H}}$ as $\{h_1, h_2, h_3\} \not\preceq \{h_1, h_2\}$. However if we declassify $\{h_2\}$ and $\{h_3\}$, resp., at points (t_1, i) and (t_2, i), for $1 \leq i \leq 4$ then the system is secure.

- $\mathcal{K}(\tau_1, i, \mathtt{L})_{\downarrow \mathtt{H}} \cap \{h_2\} \preceq \mathcal{K}(\tau_1, i+1, \mathtt{L})_{\downarrow \mathtt{H}}$
- $\mathcal{K}(\tau_2, i, \mathtt{L})_{\downarrow \mathtt{H}} \cap \{h_3\} \preceq \mathcal{K}(\tau_2, i+1, \mathtt{L})_{\downarrow \mathtt{H}}$

Security in Synchronous Systems Many definitions of information flow security have been given for synchronous systems. Such systems are represented as state machines that execute in rounds based on some logical notion of system clock. A synchronous state is a tuple $(\mathtt{HI}, \mathtt{HO}, \mathtt{LI}, \mathtt{LO})$ of high inputs, high outputs, low inputs and low outputs and not all state components are required to be present in all rounds. To handle synchronous systems, we fix an ordering of actions for each state and apply Def. 3.2.1 with synchrony parameter $t = 0$. In particular a synchronous state $(\mathtt{H}, \mathtt{L})$ of high and low actions is mapped to an asynchronous one where high actions precede low actions.

Example 3.3.5 *Consider a model of synchronous traces* $\mathcal{M} = \{(h_1, \epsilon)\bullet(\epsilon, l_1),\ (\epsilon, l_1)\bullet(h_1, \epsilon)\}$. *The model is insecure wrt.* $(\mathtt{L}, \mathtt{H})$ *as the attacker can observe the time and can deduce whether a high action has occurred. Applying the transformation, we obtain* $\mathcal{M}' = \{h_1 \bullet \epsilon \bullet \epsilon \bullet l_1,\ \epsilon \bullet l_1 \bullet h_1 \bullet \epsilon\}$, *which is still insecure (for* $t = 0$*).*

Similarly, $\mathcal{M} = \{(h_1, l_1) \bullet (h_2, l_2),\ (h_2, l_2) \bullet (h_1, l_1)\}$ *is insecure wrt.* $(\mathtt{L}, \mathtt{H})$ *since the attacker can distinguish between* h_1 *and* h_2*. As above, the model* $\mathcal{M}' = \{h_1 \bullet l_1 \bullet h_2 \bullet l_2,\ h_2 \bullet l_2 \bullet h_1 \bullet l_1\}$ *does not satisfy Def. 3.2.1.*

3.4 Equivalences

In this section we show equivalences between knowledge-based conditions and trace-based conditions from Sect. 3.2. The first proposition shows that Sep for asynchronous systems is equivalent to knowledge-based condition with security policy $\mathcal{P}ol = (\mathtt{L}, Seq(\mathtt{H}))$.

Proposition 3.4.1 *Let* $\mathcal{M}$ *be the model of an asynchronous program and closed under interleavings of* $\tau_{\mathtt{L}}$ *and* $\tau'_{\mathtt{H}}$*. Then* $\mathcal{M}$ *satisfies Sep iff* $\mathcal{M}$ *is secure wrt.* $(\mathtt{L}, Seq(\mathtt{H}))$.

Proof. We show that $\mathcal{M}$ satisfies Sep iff for all traces $\tau \in \mathcal{M},\ 0 \leq i < |\tau|$, $\mathcal{K}(\tau, i, \mathtt{L})_{Seq(\mathtt{H})} \subseteq \mathcal{K}(\tau, i+1, \mathtt{L})_{Seq(\mathtt{H})}$.

($\Rightarrow$) Suppose $\mathcal{M}$ satisfies Sep. By definition $\forall \tau_1, \tau_2 \in \mathcal{M}$ and $\forall \tau^* \in \ltimes(\tau_{1\mathtt{L}}, \tau_{2\mathtt{H}})$, $\tau^* \in \mathcal{M}$. We show that for all $\tau \in \mathcal{M}$, for all $0 \leq i < |\tau|$, $\mathcal{K}(\tau, i, \mathtt{L})_{\downarrow Seq(\mathtt{H})} \subseteq \mathcal{K}(\tau, i+1, \mathtt{L})_{\downarrow Seq(\mathtt{H})}$. Consider a sequence s^* such that $s^* \in \mathcal{K}(\tau, i, \mathtt{L})_{\downarrow Seq(\mathtt{H})}$ and show that $s^* \in \mathcal{K}(\tau, i+1, \mathtt{L})_{\downarrow Seq(\mathtt{H})}$. Let $\tau^* \in \mathcal{M}$ be such that $\tau^*_{\downarrow Seq(\mathtt{H})} = s^*$ and $(\tau^*, j) =_{\downarrow \mathtt{L}} (\tau, i)$ and $0 \leq j < |\tau^*|$. We look for a trace $\tau' \in \mathcal{M}$ with $\tau'_{\ltimes(\mathtt{H})} = s^*$ and $(\tau, i+1) =_{\downarrow \mathtt{L}} (\tau, j')$, for some j'. If event $\tau(i+1) \in \mathtt{H}$, we pick τ^* and conclude the proof. Otherwise, $\tau(i+1) \in \mathtt{L}$. Since Sep holds, it is possible to interleave the low sequence of events up to point $(\tau, i+1)$ with s^* and obtain a trace $\tau'' \in \mathcal{M}$ such that $s^* \in \mathcal{K}(\tau, i+1, \mathtt{L})_{\downarrow Seq(\mathtt{H})}$ and $(\tau, i+1) =_{\downarrow \mathtt{L}} (\tau'', j'')$, for some j''.

($\Leftarrow$) Let $\mathcal{M}$ be secure wrt. policy $\mathcal{P}ol = (\mathtt{L}, Seq(\mathtt{H}))$. By definition, $\forall \tau \in \mathcal{M},\ 0 \leq i < |\tau|$, $\mathcal{K}(\tau, i, \mathtt{L})_{\downarrow Seq(\mathtt{H})} \subseteq \mathcal{K}(\tau, i+1, \mathtt{L})_{\downarrow Seq(\mathtt{H})}$. We show that $\forall \tau_1, \tau_2 \in \mathcal{M}$ and $\forall \tau^* \in \ltimes(\tau_{1\mathtt{L}}, \tau_{2\mathtt{H}})$, $\tau^* \in \mathcal{M}$. First notice that $\mathcal{K}(\tau, 0, \mathtt{L}) = \mathcal{M}$ and by transitivity of $\preceq$ (set inclusion) it holds that $\mathcal{M}_{\downarrow Seq(\mathtt{H})} \preceq \mathcal{K}(\tau, |\tau|, \mathtt{L})_{\downarrow Seq(\mathtt{H})}$ for all $\tau \in \mathcal{M}$. This only allows to protect sequences of high events, *present in the initial model* $\mathcal{M}$.

As explained in [128], assuming a strong form of asynchrony, i.e., closure under interleavings, implies the claim immediately. □

At this point the reader may wonder if a stronger policy such as $(\mathtt{L}, Seq(\mathtt{H}\ \bot))$ can avoid the assumption of closure under interleavings. The example shows that this is not the case. The main reason is that while separability allows observers to make deductions about future or past occurrences of actions, this is not possible for the policy $(\mathtt{L}, Seq(\mathtt{H}\ \bot))$, which protects all interleavings. On the other hand, if $\mathcal{M}$ lacks some interleavings between sequences of high and low actions and this doesn't affect the initial knowledge of the observer, then the system is considered secure, whilst Sep can still break.

Example 3.4.1 *The model* $\mathcal{M} = \{h_1 \bullet h_2 \bullet l_1 \bullet l_2,\ h_1 \bullet h_2 \bullet l_1 \bullet l_3\}$ *does not satisfy Sep, but it is secure wrt. the policy* $(\mathtt{L}, Seq(\mathtt{H}\ \bot))$ *as the observer knows, from knowledge of the initial model, all interleavings, i.e,* $\{h_1 \bullet h_2 \bullet \bot \bullet \bot\}$. *On the other hand, if* $\mathcal{M} = \{h_1 \bullet l_1, l_1 \bullet h_1, h_1 \bullet l_2 \bullet l_3, l_2 \bullet h_1 \bullet l_3, l_2 \bullet l_3 \bullet h_1\}$ *and* $\mathcal{P}ol = (\mathtt{L}, Seq(\mathtt{H}\ \bot))$, *then Sep holds. However the epistemic condition does not hold since the sequence* $h_1 \bullet \bot \bullet \bot$ *is not possible after observing* l_1.

The next proposition shows that security wrt. policy $\mathcal{P}ol_1 = (\mathtt{L}, Seq(\mathtt{H}))$ is equivalent to security wrt. $\mathcal{P}ol_2 = (\mathtt{H}, Seq(\mathtt{L}))$.

Proposition 3.4.2 *A model* $\mathcal{M}$ *is secure wrt.* $(\mathtt{L}, Seq(\mathtt{H}))$ *iff* $\mathcal{M}$ *is secure wrt.* $(\mathtt{H}, Seq(\mathtt{L}))$.

Proof. We show that for all points (τ, i), $\mathcal{K}(\tau, i, \mathtt{L})_{Seq(\mathtt{H})} \subseteq \mathcal{K}(\tau, i+1, \mathtt{L})_{Seq(\mathtt{H})}$ iff $\mathcal{K}(\tau, i, \mathtt{H})_{Seq(\mathtt{L})} \subseteq \mathcal{K}(\tau, i+1, \mathtt{H})_{Seq(\mathtt{L})}$.

($\Rightarrow$) Consider $s^* \in \mathcal{K}(\tau, i, \mathtt{H})_{Seq(\mathtt{L})}$, we show that $s^* \in \mathcal{K}(\tau, i+1, \mathtt{H})_{Seq(\mathtt{L})}$. Let $\tau^* \in \mathcal{M}$ such that $\tau^*_{\downarrow \mathtt{L}} = s^*$ and $(\tau^*, j) =_{\downarrow \mathtt{H}} (\tau, i)$. If there exists $\tau' \in \mathcal{M}$ such that $\tau'_{\mathtt{L}} = s^*$ and $(\tau', j') =_{\downarrow \mathtt{H}} (\tau, i+1)$ we are done. If the event $\tau(i+1) \in \mathtt{L}$, then we are also done since τ^* is the required trace. Otherwise, $\tau(i+1) \in \mathtt{H}$. But then τ' must exists, since otherwise there would exist a sequence of high events s with prefix $(\tau, i+1)_{\mathtt{H}}$, i.e., the sequence of high events up point $(\tau, i+1)$, which is possible for $\tau_{\downarrow \mathtt{L}}$ and impossible for $\tau^*_{\downarrow \mathtt{L}}$. But then the assumption $\mathcal{K}(\tau, i, \mathtt{L})_{Seq(\mathtt{H})} \subseteq \mathcal{K}(\tau, i+1, \mathtt{L})_{Seq(\mathtt{H})}$ is false and τ' must exist. The other direction is similar. □

The next proposition shows the equivalence between Sep and its epistemic sibling in a synchronous setting.

Proposition 3.4.3 *Let* $\mathcal{M}$ *be the model of a synchronous program. Then* $\mathcal{M}$ *satisfies SSep iff* $\mathcal{M}$ *is secure wrt.* $(\mathtt{L}, Seq(\mathtt{H}))$.

Proof. We show that $\mathcal{M}$ satisfies SSep iff $\forall \tau \in \mathcal{M},\ 0 \leq i < |\tau|$, $\mathcal{K}(\tau, i, \mathtt{L})_{\downarrow Seq(\mathtt{H})} \subseteq \mathcal{K}(\tau, i+1, \mathtt{L})_{\downarrow Seq(\mathtt{H})}$. Due to the synchrony assumption, the observer knows the time, hence the knowledge at point (τ, i) is defined as $\mathcal{K}(\tau, i, \mathtt{L}) = \{\tau' | (\tau, i) =_{\downarrow \mathtt{L}} (\tau', i)\}$ since $t = 0$.

($\Rightarrow$) Suppose $\mathcal{M}$ satisfies SSep. By definition $\forall \tau_1, \tau_2 \in \mathcal{M}$, $\exists \tau^* \in \mathcal{M}$ such that $\tau^* =_{\downarrow \mathtt{L}} \tau_1$ and $\tau^* =_{\downarrow \mathtt{H}} \tau_2$. We show that for all (τ, i), $\mathcal{K}(\tau, i, \mathtt{L})_{\downarrow Seq(\mathtt{H})} \subseteq \mathcal{K}(\tau, i+1, \mathtt{L})_{\downarrow Seq(\mathtt{H})}$. By synchrony assumption, for all points i, $(\tau^*, i) =_{\downarrow \mathtt{L}} (\tau_1, i)$ and $(\tau^*, i) =_{\downarrow \mathtt{H}} (\tau_2, i)$. Then the claim follows immediately.
($\Leftarrow$) Suppose now $\mathcal{M}$ is secure wrt. $\mathcal{P}ol = (\mathtt{L}, Seq(\mathtt{H}))$. By definition for all (τ, i), $\mathcal{K}(\tau, i, \mathtt{L})_{\downarrow Seq(\mathtt{H})} \subseteq \mathcal{K}(\tau, i+1, \mathtt{L})_{\downarrow Seq(\mathtt{H})}$. We show that for all $\tau_1, \tau_2 \in \mathcal{M}$, there is a $\tau^* \in \mathcal{M}$ with $\tau^* =_{\downarrow \mathtt{L}} \tau_1$ and $\tau^* =_{\downarrow \mathtt{H}} \tau_2$. By assumption and transitivity of set inclusion, $\mathcal{K}(\tau, 0, \mathtt{L})_{\downarrow Seq(\mathtt{H})} = \mathcal{M}_{\downarrow Seq(\mathtt{H})}$ can be interleaved with all low traces in $\mathcal{M}$, since $\mathcal{M}_{\downarrow Seq(\mathtt{H})} \preceq \mathcal{K}(\tau, |\tau|, \mathtt{L})_{\downarrow Seq(\mathtt{H})}$. Then the claim follows.

□

It is worth noting that, differently from [128], no additional property on model $\mathcal{M}$ is needed to show the equivalence in Prop. 3.4.3. The main reason is that our work considers traces of both finite and infinite length, hence no property as *limit closure* is required.

Proposition 3.4.4 *Let $\mathcal{M}$ be closed under interleavings of $\tau_{\mathtt{L}}$ and $\tau'_{\mathtt{HI}}$. Then $\mathcal{M}$ satisfies GNI iff $\mathcal{M}$ is secure wrt. $(\mathtt{L}, Seq(\mathtt{HI}))$.*

Proof. Similar to Prop. 3.4.1. □

Proposition 3.4.5 *A model $\mathcal{M}$ satisfies ND iff $\mathcal{M}$ is secure wrt. $(\mathtt{L}, Seq(\mathtt{HI}))$.*

Proof. Similar to Prop. 3.4.4, but no closure under interleavings is assumed. □

Proposition 3.4.6 *If a model $\mathcal{M}$ is secure wrt. $(\mathtt{L}, Seq(\mathtt{H}))$ then $\mathcal{M}$ satisfies NDS.*

Proof. Suppose towards contradiction that s_1 and s_2 are two strategies such that there exist $\tau, \tau' \in \mathcal{M}$ with $s_1(\tau_{\mathtt{L}}) = s_2(\tau'_{\mathtt{L}})$ and $\tau_{\mathtt{L}} \neq \tau'_{\mathtt{L}}$. This implies that there exists a sequence of high events, say $\tau_{\mathtt{H}}$, which is not possible when observing low sequence $\tau'_{\mathtt{L}}$. But this would violate the epistemic condition which guarantees that any H sequences of model is possible for any L observation. Hence, the claim follows.
□

The following example shows that the security condition is strong enough to capture information flows through high user strategies.

Example 3.4.2 *Consider program P from [224] where a high user transmits a bit z to a low user by sending $z \otimes x$ as input on high channel. Let $c_1 \in \mathcal{O}$ and $c_2, c_3 \in \mathcal{P}$. Then what the low user receives is the exact value of secret z.*

$$P ::= x := 0 || 1;\ \mathbf{out}(c_3, x);\ \mathbf{in}(c_2, y);\ \mathbf{out}(c_1, x \otimes y)$$

It can be checked that M_P is secure wrt. $(\mathtt{L}, Seq(\mathtt{HI}))$, i.e. ND, and insecure wrt. $(\mathtt{L}, Seq(\mathtt{H}))$. Hence, P does not satisfy NDS.

The following propositions show that the epistemic conditions can be seen as closures over a poset where the ordering relation is given by the security policy. This gives a systematic characterization of security conditions wrt. what is protected and how powerful an attacker is.

Proposition 3.4.7 *The security policies $\mathcal{P}ol = (\mathcal{O}, \mathcal{P})$ are closures over a poset $(\wp(E^*), \preceq)$, where E^* is any sequence of events. In particular,* $(\mathtt{L}, \mathtt{H}) \sqsubseteq (\mathtt{L}, \mathit{Mul}(\mathtt{H})) \sqsubseteq (\mathtt{L}, \mathit{Seq}(\mathtt{H})) \sqsubseteq (\mathtt{L}, \mathit{Seq}(\mathtt{H}\ \bot))$.

Proof. Let $\mathcal{M}$ be a model and $(\wp(\mathcal{M}), \preceq)$ a poset ordered by subset inclusion. Then each policy can be associated with an *uco* over $\wp(\mathcal{M})$. Consider a non empty set $S \in \wp(\mathcal{M})$. Then for all above policies $\mathcal{P}ol = (\mathcal{O}, \mathcal{P})$, we define $\rho_{\mathcal{P}ol}(S) = \mathcal{M}$ if there exist $\tau, \tau' \in S$ and $\tau_{\downarrow\mathcal{O}}$ is different from $\tau'_{\downarrow\mathcal{O}}$. Moreover, $\rho(\emptyset) = \emptyset$. Otherwise, for all subsets S having the same $\mathcal{O}$ observations, we define $\rho_{\mathcal{P}ol}(S) = \bigcup_{\tau \in S}\{\tau' \in \mathcal{M} | \tau =_{\downarrow\mathcal{P}} \tau'\}$. It can be easily proved that $\rho_{\mathcal{P}ol}$ is an uco. Finally, the order relation, $\rho_1 \sqsubseteq \rho_2$ iff for all S, $\rho_1(S) \supseteq \rho_2(S)$, determines the ordering between closures, i.e. policies. □

Proposition 3.4.8 *Let $\mathcal{M}$ be a model and $\mathcal{P}ol_1 = (\mathcal{O}_1, \mathcal{P}_1)$, $\mathcal{P}ol_2 = (\mathcal{O}_2, \mathcal{P}_2)$ be security policies. If $\mathcal{P}ol_1 \sqsubseteq \mathcal{P}ol_2$, i.e., $\mathcal{O}_1 \sqsubseteq \mathcal{O}_2$ and $\mathcal{P}_1 \sqsubseteq \mathcal{P}_2$, then $\mathcal{M}$ is secure wrt. $\mathcal{P}ol_2$ if $\mathcal{M}$ is secure wrt. $\mathcal{P}ol_1$.*

Proof. Follows from definition of closures in Prop. 3.4.7. □

We conclude this section by showing equivalence between nondeducibility on outputs and its epistemic peer condition. The main advantage of NDO is that it allows information flow from low user input channels to high output channels and, at the same time it protects sequences of high actions interleaved with low inputs. To express such requirements, we make use of the release policy which allows a low user to declassify information about sequences of low user inputs and high actions, i.e., $\mathit{Seq}(\mathtt{LI} + \mathtt{H})$, given the low input sequence $\mathtt{LI}$ of the current trace, namely, $\mathcal{R}(\tau, i, \mathtt{LI}) = \{\tau^* \in \mathcal{M} | \tau =_{\mathtt{LI}} \tau^*\}$. The other low inputs, *system inputs* in [127], are modeled as internal nondeterminism.

Proposition 3.4.9 *Consider a program model $\mathcal{M}$. Then $\mathcal{M}$ satisfies NDO iff* $\forall \tau, i,\ \mathcal{K}(\tau, i, \mathtt{L})_{\downarrow \mathit{Seq}(\mathtt{LI}+\mathtt{H})} \cap \mathcal{R}(\tau, i, \mathtt{LI})_{\downarrow \mathit{Seq}(\mathtt{LI}+\mathtt{H})} \preceq \mathcal{K}(\tau, i+1, \mathtt{L})_{\downarrow \mathit{Seq}(\mathtt{LI}+\mathtt{H})}$.

Proof. ($\Rightarrow$) Suppose $\mathcal{M}$ satisfies NDO. Then, for all $\tau, \tau' \in \mathcal{M}$ such that $\tau_{\mathtt{LI}} = \tau'_{\mathtt{LI}}$ there exists $\tau^* \in \mathcal{M}$ and $\tau^*_{\mathtt{H}+\mathtt{LI}} = \tau_{\mathtt{H}+\mathtt{LI}}$ and $\tau^*_{\mathtt{L}} = \tau'_{\mathtt{L}}$. Consider a point (τ, i) and a sequence s^* of $\mathit{Seq}(\mathtt{LI} + \mathtt{H})$ events with $s^* \in \mathcal{K}(\tau, i, \mathtt{L})_{\downarrow \mathit{Seq}(\mathtt{LI}+\mathtt{H})}$ and $s^* \in \mathcal{R}(\tau, i, \mathtt{LI})_{\downarrow \mathit{Seq}(\mathtt{LI}+\mathtt{H})}$, we show that $s^* \in \mathcal{K}(\tau, i+1, \mathtt{L})_{\downarrow \mathit{Seq}(\mathtt{LI}+\mathtt{H})}$. By assumption, there exists $\tau^* \in \mathcal{M}$ such that $\tau^*_{\downarrow \mathit{Seq}(\mathtt{LI}+\mathtt{H})} = s^*$, $(\tau^*, j) =_{\downarrow \mathtt{L}} (\tau, i)$ for some j, and $\tau^* =_{\downarrow \mathtt{LI}} \tau$. We show there exists $\tau' \in \mathcal{M}$ such that $\tau'_{\downarrow \mathit{Seq}(\mathtt{LI}+\mathtt{H})}$ and $(\tau', j') =_{\downarrow \mathtt{L}} (\tau, i+1)$. If $\tau(i+1) \notin \mathtt{L}$ we are done, since τ^* would do. Otherwise $\tau(i+1)$ is a

low event and NDO enforces existence on τ' such that $\tau' =_{\downarrow \mathtt{L}} \tau$. In particular, for all k there exists k', $(\tau, k) =_{\downarrow \mathtt{L}} (\tau', k')$, hence $(\tau, i+1) =_{\downarrow \mathtt{L}} (\tau', j)$ and τ' exist. ($\Leftarrow$) Suppose the epistemic condition holds. The for all $\tau \in \mathcal{M}$, $\mathcal{K}(\tau, 0, \mathtt{L})_{\downarrow Seq(\mathtt{LI+H})} \cap \mathcal{R}(\tau, 0, \mathtt{LI})_{\downarrow Seq(\mathtt{LI+H})} \preceq \mathcal{K}(\tau, |\tau|, \mathtt{L})_{\downarrow Seq(\mathtt{LI+H})}$ which implies NDO immediately. □

The following example from [165], shows that our condition handles correctly information from LI to HO. The key point here is the use of release policy to break the symmetry inherent in nondeducibility-like conditions.

Example 3.4.3 *Consider P with low channels c_1, c_3 and high channels c_2, c_4.*

$$P ::= \mathbf{in}(c_1, x); \mathbf{out}(c_2, x); \mathbf{out}(c_3, x); \mathbf{in}(c_4, y)$$

If $\mathbf{in}(c_1, x)$ is a low user input, the program can be considered secure as nothing about high actions is revealed. However, if $\mathbf{in}(c_1, x)$ is a system input, then the value is incorrectly transmitted to the low user through $out(c_3, x)$. The security condition captures both cases.

3.5 A logic for Information Flow

In this section we express the security conditions in Sect. 3.2 in terms of a logic of knowledge and time. We consider the framework of multi-agent systems [112] and extend the logic presented in [38] to reason about possibilistic security conditions.

3.5.1 Knowledge in Multi-agent Systems

The framework of multi-agent systems allows reasoning about knowledge and time in a distributed system where different agents (users, processes) interact with each other. The system consists of a set of agents $Ag = \{a_i\}_{i=1}^{k}$ which have local state L_i at a given point in time. A special agent E, with local state L_E, models the environment where the distributed system runs. The global state consists of a tuple of local states, i.e., $G = (L_E, L_1, \cdots, L_k)$. A *run* is a sequence of global states over discrete time. An *interpreted system* [112] is a pair $\mathcal{I} = (R, \Pi)$ of runs R and interpretation function Π over a set Φ of atomic propositions. Program models can be associated with interpreted systems. Given a point (τ, i), then $L_E = (trace(\tau, i))$. The local state of an agent a who observes $\mathcal{O}$ is $L_a = (trace(\tau, i)_{\downarrow \mathcal{O}})$. Then, the global state of a system with k agents who observe $\mathcal{O}_1, \cdots, \mathcal{O}_k$ is $G = ((trace(\tau, i)), (trace(\tau, i)_{\downarrow \mathcal{O}_1}), \cdots, (trace(\tau, i)_{\downarrow \mathcal{O}_k}))$. The atomic propositions in Φ describe basic facts about the model. In our context, the facts refer to actions on channels. The interpretation function $\Pi(p)(G)$ assigns a truth value to all $p \in \Phi$. To define knowledge, an interpreted system $\mathcal{I} = (R, \Pi)$ is associated with a *Kripke structure* $\mathcal{M}_{\mathcal{I}} = (G, \Pi, \{K_i\}_{i=1}^{k})$ where G and Π are as before and K_i is a binary relation over G. In particular, $K_i(G) = \{G' | G \sim_i G'\}$, where $G \sim_i G'$ if L_i is the same in both states.

3.5.2 Temporal Epistemic Logic with Past

We now present a logic with temporal and epistemic operators to reason about security properties in a syntactical manner. Let $\Phi = \{\mathbf{in}(c, v), \mathbf{out}(c', v') | c, c' \in \mathit{Chan} \text{ and } v, v' \in \mathit{Val}\}$ be the set of atomic propositions. We consider a language containing Φ and closed off under conjunction, negation, knowledge operators K_a, temporal operators *Next* X and *Until* U and past time operators *Initially* I, *Previous* Y and *Since* S. Let $p \in \Phi$,

Definition 3.5.1 (Syntax of $\mathcal{L}_{KU}$)
The language $\mathcal{L}_{KU}$ of formulas ϕ, ψ in linear time temporal epistemic logic with past is given as follows:

$$\phi, \psi ::= p \mid \phi \wedge \psi \mid \neg\phi \mid K_a\phi \mid X\phi \mid \phi U \psi \mid I\phi \mid Y\phi \mid \phi S \psi$$

The operator K_a is the epistemic knowledge operator. $K_a\phi$ holds if ϕ holds in any point equivalent to the current point of agent a. The formula $\phi U \psi$ holds if ψ holds in a future point and ϕ holds until reaching that point. Dually, the formula $\phi S \psi$ holds if ψ was true once in the past and ϕ has been true ever since. $I\phi$ holds if ϕ is true initially, while $X\phi$ ($Y\phi$) hold if ϕ is true at the next (previous) point. Various connectives are definable in $\mathcal{L}_{KU}$ including boolean operators such as $\vee$ and $\rightarrow$, the truth constants *tt* and *ff*, the epistemic possibility operator $L_a\phi$ meaning that ϕ holds for at least one epistemically equivalent point, the future (past) operator $F\phi$ ($O\phi$) requiring ϕ to eventually hold in the future (past), the always (historically) operator $G\phi$ ($H\phi$) meaning that ϕ holds in any future (past) state, and the weak until $\phi W \psi$ which does not require ψ to eventually hold. Finally, the operator K_G is the group knowledge operator, the formula $K_G\phi$ holds if the combined knowledge of G members implies ϕ. The logic is sufficiently expressive to specify all information flow policies in Sect. 3.1.

Definition 3.5.2 (Satisfaction) *Fig. 3.1 defines the satisfaction relation $\mathcal{M}, (\tau, i) \models \phi$ between points in a model $\mathcal{M}$ and $\mathcal{L}_{KU}$ formulas. In particular, satisfaction relative to model $\mathcal{M}$ is defined as $\mathcal{M} \models \phi$ iff $\forall \tau \in \mathcal{M},\ \mathcal{M}, (\tau, 0) \models \phi$.*

At this point we have all ingredients to characterize the security condition in Sect. 3.2 by means of $\mathcal{L}_{KU}$ formulas. First notice that the set $\mathcal{K}(\tau, i, \mathcal{O})$ represents all traces that an observer $\mathcal{O}$ considers possible at point (τ, i), which corresponds to the uncertainty of the observer at that point. Given a policy $\mathcal{P}ol = (\mathcal{O}, \mathcal{P})$ and a formula ϕ specifying properties of $\mathcal{P}$, we show that ϕ is possible at any point (τ, i) by means of the operator $L_a\phi$. To avoid complications due to observations at the limit, we assume that the models are limit closed. Namely, all properties of $\mathcal{P}$ can be captured by finitely many observations in $\mathcal{O}$. The view of agent a (group G) observing $\mathcal{O}_a$ ($\mathcal{O}_G = \bigcup_{a \in G} \mathcal{O}_a$) is defined as $\mathit{trace}_a(\tau, i) = \mathit{trace}(\tau, i)_{\downarrow \mathcal{O}}$ ($\mathit{trace}_G(\tau, i) = \mathit{trace}(\tau, i)_{\downarrow \mathcal{O}_G}$). Then the following theorem relates the semantic conditions to the syntactical ones.

$$
\begin{array}{ll}
\mathcal{M},(\tau,i) \models p & \text{iff } \tau(i) = p \\
\mathcal{M},(\tau,i) \models \phi \wedge \psi & \text{iff } (\tau,i) \models \phi \text{ and } (\tau,i) \models \psi \\
\mathcal{M},(\tau,i) \models \neg\phi & \text{iff } (\tau,i) \not\models \phi \\
\mathcal{M},(\tau,i) \models X\phi & \text{iff } i+1 \leq len(\pi) \text{ and } (\tau,i+1) \models \phi \\
\mathcal{M},(\tau,i) \models \phi U \psi & \text{iff } \exists j : i \leq j \leq len(\tau) \text{ such that } (\tau,j) \models \psi \\
 & \quad \text{and } \forall k : i \leq k < j,\ (\tau,k) \models \phi \\
\mathcal{M},(\tau,i) \models I\phi & \text{iff } (\tau,0) \models \phi \\
\mathcal{M},(\tau,i) \models Y\phi & \text{iff } i-1 \geq 0 \text{ and } (\tau,i-1) \models \phi \\
\mathcal{M},(\tau,i) \models \phi S \psi & \text{iff } \exists j : 0 \leq j \leq i \text{ such that } (\tau,j) \models \psi \\
 & \quad \text{and } \forall k : j < k \leq i,\ (\tau,k) \models \phi \\
\mathcal{M},(\tau,i) \models K_a\phi & \text{iff } \forall \tau' \in \mathcal{M}, \forall (\tau',i') \in \tau' \text{ such that} \\
 & \quad trace_a(\tau,i) = trace_a(\tau',i'),\ (\tau',i') \models \phi
\end{array}
$$

Figure 3.1: Satisfaction at trace points

Theorem 3.5.1 *Consider a model $\mathcal{M}$ and a security policy $\mathcal{P}ol = (\mathcal{O},\mathcal{P})$. Let also $\phi_1,\cdots,\phi_n$ be a set of $\mathcal{L}_{KU}$ formulas encoding information to be protected, i.e. $\mathcal{M}_{\downarrow\mathcal{P}}$. Then $\mathcal{M}$ is secure wrt. $\mathcal{P}ol$ iff $\mathcal{M} \models \bigwedge_{i=1}^{n} GL_a\phi_i$.*

Proof. We show that for all $\tau \in \mathcal{M}$ and $0 \leq i < |\tau|$, $\mathcal{K}(\tau,i,\mathcal{O})_{\downarrow\mathcal{P}} \preceq \mathcal{K}(\tau,i+1,\mathcal{O})_{\downarrow\mathcal{P}}$ iff for all $\tau \in \mathcal{M}$, $(\tau,0) \models \bigwedge_{i=1}^{n} GL_a\phi_i$.

($\Rightarrow$) Consider a formula ϕ_i which encodes an element in $\mathcal{P}$. Following the rules in Fig. 3.1, we show that $(\tau,i) \models L_a\phi_i$. By assumption, $\mathcal{M}_{\downarrow\mathcal{P}} \preceq \mathcal{K}(\tau,i,\mathcal{O})_{\downarrow\mathcal{P}}$ for all (τ,i), and in particular ϕ_i. Then, there exists $\tau' \in \mathcal{K}(\tau,i,\mathcal{O})$ and $\tau'_{\downarrow\mathcal{P}} = \phi_i$. Hence, there exists $(\tau',i') =_{\downarrow\mathcal{O}} (\tau,i)$ and $\tau'_{\downarrow\mathcal{P}} = \phi_i$, which implies $(\tau',i') \models \phi_i$ and, consequently, $(\tau,i) \models L_a\phi_i$.

($\Leftarrow$) Conversely, suppose that $\mathcal{M} \models GL_a\phi_i$, for all ϕ_i which encodes $\mathcal{M}_{\downarrow\mathcal{P}}$. Then for all (τ,i), $\tau,i \models L_a\phi_i$. Namely, there exists (τ',i') such that $(\tau,i) =_{\downarrow\mathcal{O}} (\tau',i')$ and $(\tau',i') \models \phi_i$, for all ϕ_i. Consequently, for all (τ,i), $\mathcal{M}_{\downarrow\mathcal{P}} \preceq \mathcal{K}(\tau,i,\mathcal{O})_{\downarrow\mathcal{P}}$. This implies that $\mathcal{K}(\tau,i,\mathcal{O})_{\downarrow\mathcal{P}} \preceq \mathcal{K}(\tau,i+1,\mathcal{O})_{\downarrow\mathcal{P}}$. □

Finally it remains to show that the logic can be used describe different protection policies, as defined in Sect. 3.3. In particular, each element in $\mathcal{M}_{\downarrow\mathcal{P}}$ can be encoded using the logic in Fig. 3.1. Let $p \in \Phi$, then we define auxiliary formulas: $Occ(p) = (O\ p \vee F\ p)$ meaning that p eventually holds at a (past or future) point, $SV(c) = \bigvee_{v \in Val} e(c,v)$ meaning that an action has happened on channel c, $Occ(p,i) = O\ (I\ tt\ \wedge F\ (p \wedge X\ F\ (p \wedge X\ F\ (p \wedge \cdots))))$ meaning that p is true in at least i different points in the current trace and a happens-before formula $HB(p,q) = O\ (\neg q U (p \wedge F\ q)) \vee F\ (\neg q U (p \wedge F\ q))$ meaning that p holds before q at some point

in the current trace. Moreover, auxiliary formulas can be combined to express facts ϕ that occur infinitely many times by using the formula $Inf(\phi) = G\,F\phi$.

Proposition 3.5.1 *Consider a model $\mathcal{M}$ and a policy $\mathcal{P}ol = (\mathcal{O}, \mathcal{P})$. Then the following variants of $\mathcal{P}$ can be encoded in $\mathcal{L}_{KU}$,*

- *Set*(H)*: $\mathcal{M} \models \bigwedge_{i=1}^{n} GL_a\, Occ(e_i(c_i, v_i))$, where $c_i \in \mathcal{P}$*
- *Mul*(H)*: $\mathcal{M} \models \bigwedge_{i=1}^{n} GL_a\, Occ(e_i(c_i, v_i), k_i)$, where $c_i \in \mathcal{P}$ and k_i is the multiplicity of e_i*
- *Seq*(H)*: $\mathcal{M} \models L_a \bigwedge_{j=2}^{k} HB(e_{j-1}, e_j)$ for all high sequences ϕ_i*
- *Seq*(H $\perp$) *iff $\mathcal{M} \models GL_a \bigwedge_{j=2}^{k} HB(p_{j-1}, p_j)$, for all sequences ϕ_i such that $p_j = SV(c)$ if $e_j(c, v) = p_j$ and $c \in \mathcal{O}$, or $p_j = e_j(c, v)$ if $c \in \mathcal{P}$*

Proof. Direct check on satisfaction relation in Fig. 3.1. □

3.6 Related Work and Conclusions

We are not the first to examine connection between trace-based and knowledge-based information flow properties. A closely related work is that of Halpern and O'Neill [128], who also consider formal definitions of secrecy in multiagent systems. They show how SEP and GNI fit in the epistemic framework, both in synchronous and asynchronous setting. In particular, they define knowledge as a set of points that an agent considers possible based on his local state. By contrast, this paper defines knowledge as the set of global traces that an agent considers possible based on his local observations. This allows us to give security conditions which are closer to what is used in language-based security [27, 24, 37]. Furthermore, the logic we present here captures directly the security properties of traces.

Recently, knowledge-based conditions for information flow have been popular in language-based security. Several works [182, 29] explore epistemic conditions for relational and interactive models, although in a synchronous setting only. Different issues related to declassification [38] and attack models [24] have been considered using epistemic security conditions. In [196] Sabelfeld and Mantel discuss information flow security for distributed programs and point out finer-grained sources of leaks due to encryption, environment totality and timing. All these subtle flows can be accurately captured by the security condition presented here.

The majority of verification techniques for information flow properties rely on security type systems, as in [218]. However, several model checking approaches, which were recently proposed [19, 104], define fragments of logics for which verification is feasible. An interesting future direction would be to devise fragments of $\mathcal{L}_{KU}$ for which model checking has low complexity.

In conclusion, we have discussed several possibilistic information flow conditions and showed how knowledge-based account can be used to specify these conditions, both semantically and syntactically. The advantage of using epistemic logic is that it can accurately express complex policies and provide a clear separation between the code and the security annotations. However, complexity of verification can be very high for large programs. Different remedies to this issue require further investigation. First, abstraction techniques at the program level can be used to obtain smaller models which can be easy to verify. Second, distributed system properties, such as asynchrony, order preservation or lossiness can be used to decompose the epistemic formulas into simpler ones, which are easier to verify. Finally, a hybrid verification which combines type checking and model checking is another path that deserves further exploration.

Acknowledgements. Thanks to Mads Dam for many valuable discussions. This work was supported by SSF-funded project PROSPER (№ RIT10-0069).

Chapter 4

A Weakest Precondition Approach to Robustness

Musard Balliu and Isabella Mastroeni

Abstract

With the increasing complexity of information management computer systems, security becomes a real concern. E-government, web-based financial transactions or military and health care information systems are only a few examples where large amount of information resides on different hosts distributed worldwide. It is clear that any disclosure or corruption of sensitive information in these contexts may result fatal. Information flow control constitutes an appealing and promising technology to protect both data confidentiality and data integrity. The certification of the security degree of a program that runs in untrusted environments still remains an open problem in the area of language-based security. Robustness asserts that an active attacker, who can modify the program code in some fixed program points, is unable to disclose more sensitive information than a passive attacker, who merely observes the public data. In this paper, we extend a method recently proposed for checking declassified noninterference in presence of passive attackers only, in order to check robustness by means of the weakest precondition semantics. The weakest precondition semantics simulates the kind of analysis that can be performed by an attacker, i.e. from public output towards private input. The choice of the semantics allows us to distinguish between different types of attack models and thus characterize the security of programs in different scenarios. Our results can be used to address confidentiality and integrity of software running in untrusted environments where different agents can distrust one another. For instance, a web server can be attacked in order to steal a session cookie or to hijack clients to a fake web page.

4.1 Introduction

Security is an enabling technology, hence security means power. For example, the correct functionality and coordination of large scale organizations, e-government and web services in general relies on the confidentiality and the integrity of the data exchanged between different participating agents. Nowadays, distributed and service-oriented architectures are the first business alternative to the old fashioned client-server architectures. According to OWASP (Open Web Application Security Project) [10], the most critical security failures are due to application level attacks such as SQL injection or XSS (Cross Site Scripting). Moreover, current and future trends in software engineering prognosticate mobile code technology (multi application smart cards, software for embedded systems), extensibility and platform independence. It is worth noting that all these features, almost unavoidable, become real opportunities for the attackers to exploit system bugs in order to disclose and/or corrupt valuable information. This makes it easier to distribute worms or viruses that run everywhere or to embed malicious code that exploits vulnerabilities in a web server.

In many scenarios, different agents, each having their own security policy and probably not trusting each other, have to cooperate to achieve a certain goal, for example electing the winner in an online auction. It can happen that the host used for the computation violates security by either leaking information itself or causing other hosts to leak information [228, 74]. In a cryptographic context, secure multiparty computation [225] aims at computing a function between different agents, each knowing a secret they don't want to reveal to the other participating agents. This are all examples where the attacker can possibly be part of the system and, by taking the control of some host, she may reveal the private data of the other participating hosts. As a result, it is both useful and necessary to address issues regarding the confidential information disclosed by an attacker that controls part of the system. This can be done by characterizing the possible harm caused when some condition is verified or by stating sufficient conditions that guarantee robustness. Application level verification, which combines programming languages and static analysis, has different tools to tackle this problem [227, 74].

Secure information flow concerns the problem of protecting private information from an untrusted observer. This problem is indeed actual each time a program, manipulating both sensitive and public information, is executed in an untrusted environment. In this case, security is usually enforced by means of *noninterference* policies [122], stating that private information must not affect the observable public information. In the noninterference context, variables have a confidentiality level, usually public/low and private/high, and any variation of the private input should not affect the public output. This holds when considering attackers that can only observe the I/O behavior of the program and that, from these observations, can make some kind of *reverse engineering* to derive the private information that the program may leak.

Our starting idea is that of finding the program vulnerabilities by simulating

the possible reasoning that an attacker can perform on the program. Indeed, the attacker can use the output observation in order to derive, *backwards*, the (even partial) private input information. This is the idea of the *backward analysis* recently proposed in [42] for declassified noninterference, where declassification is modeled by means of abstract domains [89]. The ingredients of this method are: the initial declassification policy modeled as an abstraction of the private input domain and the weakest (liberal) precondition semantics (*Wlp*) of program [103, 102], characterizing the backward analysis (i.e. from outputs to inputs) and the simulation of the attacker's observational activity. The certification process consists in considering a possible (public) output observation and computing the *Wlp* of the program starting from this observation. By definition, the *weakest* precondition semantics provides the *greatest* set of possible input states leading to the given output observation. In other words, it characterizes the greatest collection of input states, and in particular of private inputs, that an attacker can identify starting from the given observation. In this way, the attacker can restrict the range of private inputs inside this collection, which may correspond to a release of private information. Moreover, the fact that we compute the *weakest* precondition for the given observation, provides a characterisation of the *maximal* information released by that observation, in the lattice of abstract interpretations. Namely, starting from the results provided by the analysis, we construct an abstract domain, representing the private abstract property that is released, which corresponds to the most concrete property released by the program [42].

Our aim is to use these ideas also in presence of *active attackers*, namely attackers that can both observe and modify the program semantics. We consider the model of active attackers proposed in [176], where the attacker can transform the program semantics simply by inserting malicious code in fixed program points (*holes*), known by the programmer. We show that, also in presence of this kind of attacker, the *Wlp* computation can be used to characterize the disclosed information, and therefore to spot program vulnerabilities. This characterisation can be interpreted from two opposite points of view: the attacker and the program administrator. The attacker can be any malicious agent trying to disclose confidential information about the system; the administrator wants to know whether the system releases private information due to particular inputs.

As mentioned earlier, an important security property concerning active attackers, and related to the disclosed information, is robustness [227]. Robustness "measures" the security degree of the program wrt. an active attacker by certifying that active attacker cannot disclose more information than what a passive attacker (a simple observer) can do. We propose to use the *Wlp* analysis to certify program robustness. The first idea we consider is to compute the maximal information disclosed both for passive attackers [42] and for active attackers, and then compare the results in the lattice of abstract interpretations: if there exists at least one active attacker disclosing more than the passive one, the program fails to be robust. The problem with this technique is that it requires a program analysis for each attack. This means that it becomes unfeasible when dealing with an infinite number of

possible active attacks. To overcome this problem, we need an analysis independent of the code of the particular active attack. For this reason, we leverage the weakest precondition computation and provide a *sufficient condition* that guarantees robustness independently of the attack. In particular we provide a condition that has to hold before each hole, for preventing the attacker to be successful. We initially study this condition for I/O attackers, i.e. attackers that can only observe the public I/O program behavior, and then extend it to attackers able to observe also intermediate states, i.e. the trace semantics of the program. Finally, we note that, in some restricted contexts, for example where the activity of the attacker is limited by the environment, the standard notion of robustness may become too strong. For dealing with these situations we introduce a weakening of robustness, i.e. *relative robustness*, where we restrict the set of active attackers that we are checking robustness for.

There are various interesting applications where our approach can be used to analyse confidential information flows. Here we present two case studies concerning API security and XSS attacks and apply the *Wlp* analysis to check robustness. The first case study considers the security of an API used to verify the password inserted in an ATM cash machine. The adversary is able to reveal the entire password by tampering with low integrity data prior to the API function call [70]. The second example concerns a web attack using Javascript. As we will see, a naive control of code integrity can reveal the entire session cookie to the attacker [179, 79]. The robustness analysis we propose is sufficient to reveal the attacks in both examples.

Roadmap. The rest of the paper is organized as follows. In Section 4.2 we give a general overview of abstract interpretation, which constitutes the underlying framework that we use to compare the information. In Section 4.3 we present the security background needed to understand our approach. In particular we recall notions of noninterference, robustness, declassification, decentralized label model and decentralized robustness. In Section 4.4 we compute (qualitatively) the maximal private information disclosed by an active attacker. In particular, Section 4.4.1 introduces the problem of computing the maximal release by an active attacker for I/O (denotational) semantics. Section 4.4.2 extends the analysis to an attacker observing the program traces. In Section 4.5 we discuss several conditions to enforce robustness, which is our main contribution. Section 4.5.1 presents the static analysis approach based on weakest preconditions to enforce robustness for I/O semantics; Section 4.5.3 extends these results for trace semantics; In Section 4.5.4 we compare our method with type-based approaches. Section 4.6 introduces relative robustness which deals with restricted classes of attacks; in Section 4.6.1 we interpret decentralized robustness in our model. In Section 4.7 we exercise our approach in the context of real applications and explain how it captures the security properties we are interested in. Section 4.8 presents the related work. We conclude with Section 4.9 by discussing the current state of art and new directions for future work. This is an extended and revised version of [40].

4.2 Abstract Interpretation: An Informal Introduction

We use the standard framework of abstract interpretation [89, 90] for modeling properties. For example, instead of computing on integers we might compute on more abstract properties, such as the sign $\{+, -, 0\}$ or the parity $\{\texttt{even}, \texttt{odd}\}$. Consider the program $\texttt{sum}(x, y) = x + y$, then it is abstractly interpreted as $\texttt{sum}^*$: $\texttt{sum}^*(+, +) = +$, $\texttt{sum}^*(-, -) = -$, but $\texttt{sum}^*(+, -) =$ *"I don't know"* since we are not able to determine the sign of the sum of a negative number with a positive one (modeled by the fact that the result can be any value). Analogously, $\texttt{sum}^*(\texttt{even}, \texttt{even}) = \texttt{even}$, $\texttt{sum}^*(\texttt{odd}, \texttt{odd}) = \texttt{even}$ and $\texttt{sum}^*(\texttt{even}, \texttt{odd}) = \texttt{odd}$. More formally, given a concrete domain C we may chose to describe the abstractions of C as upper closure operators. An *upper closure operator* (uco for short) $\rho : C \to C$ on a poset C is monotone, idempotent, and extensive: $\forall x \in C.\ x \leq_C \rho(x)$. The upper closure operator is the function that maps the concrete values with their abstract properties, namely with the best possible approximation of the concrete value in the abstract domain. For example, the operator $\mathit{Sign} : \wp(\mathbb{Z}) \to \wp(\mathbb{Z})$, on the powerset of integers, associates each set of integers S with its sign: $\mathit{Sign}(\varnothing) =$*"none"*, $\mathit{Sign}(S) = +$ if $\forall n \in S.\ x > 0$, $\mathit{Sign}(0) = 0$, $\mathit{Sign}(S) = -$ if $\forall n \in S.\ n < 0$ and $\mathit{Sign}(S) =$ *"I don't know"* otherwise. The property names *"none"*, +,0,− and *"I don't know"* are the names of the following sets in $\wp(\mathbb{Z})$: $\varnothing$, $\left\{\ n \in \mathbb{Z} \,\middle|\ n > 0\ \right\}$, $\{0\}$, $\left\{\ n \in \mathbb{Z} \,\middle|\ n < 0\ \right\}$ and $\mathbb{Z}$. Namely the abstract elements, in general, correspond to the set of values with the property they represent. Analogously, we can define an operator $\mathit{Par} : \wp(\mathbb{Z}) \to \wp(\mathbb{Z})$ associating each set of integers with its parity. $\mathit{Par}(\varnothing) =$ *"none"* $= \varnothing$, $\mathit{Par}(S) = \texttt{even} = \left\{\ n \in \mathbb{Z} \,\middle|\ n \text{ is even}\ \right\}$ if $\forall n \in S.\ n$ is even, $\mathit{Par}(S) = \texttt{odd} = \left\{\ n \in \mathbb{Z} \,\middle|\ n \text{ is odd}\ \right\}$ if $\forall n \in S.\ n$ is odd and $\mathit{Par}(S) =$ *"I don't know"* $= \mathbb{Z}$ otherwise. Formally, closure operators ρ are uniquely determined by the set of their fix-points $\rho(C)$, for instance $\mathit{Sign} = \{\mathbb{Z}, > 0, < 0, 0, \varnothing\}$. Abstract domains on the complete lattice $\langle C, \leq, \vee, \wedge, \top, \bot\rangle$ form a complete lattice, formally denoted $\langle \mathit{uco}(C), \sqsubseteq, \sqcup, \sqcap, \lambda x.\ \top, \lambda x.\ x\rangle$, where $\rho \sqsubseteq \eta$ means that ρ is more concrete than η, namely it is more precise, $\sqcap_i \rho_i$ is the greatest lower bound taking the most abstract domain containing all the ρ_i, $\sqcup_i \rho_i$ is the least upper bound taking the most concrete domain contained in all the ρ_i, $\lambda x.\ \top$ is the most abstract domain unable to distinguish concrete elements, the identity on C, $\lambda x.\ x$, i.e. the concrete domain, is the most concrete abstract domain.

4.3 Security Background

Information flow models of confidentiality, also called noninterference models [122], are widely studied in the literature [197]. Generally they consider the denotational semantics of a program P, denoted as $[\![P]\!]$ and all program variables, in addition to their base type (e.g. int, float), have a security type that varies between private/high (H) and public/low (L). In this paper we consider only terminating computations. Hence, there are basically two ways the program can release private

information by the observation of the public outputs: due to an explicit flow corresponding to a direct assignment of a private variable to a public variable and due to an implicit flow corresponding to control structures of the program, such as the conditional `if` or the `while` loop [197].

4.3.1 Noninterference and Declassification

A program satisfies *(standard) noninterference* if for all variations of the private input data there is no variation of the public output data. More formally, given a set of program states Σ, namely a set of functions mapping program variables to values $\mathbb{V}$, we represent a state as a tuple $(\vec{h}, \vec{l})$, where the first component denotes the value of the private variables and the second component denotes the value of the public variables. Let P be a program, then P satisfies standard noninterference if

$$\forall l \in \mathbb{V}^{\mathtt{L}}, \forall h_1, h_2 \in \mathbb{V}^{\mathtt{H}}. [\![P]\!](h_1, l)^{\mathtt{L}} = [\![P]\!](h_2, l)^{\mathtt{L}}$$

where $v \in \mathbb{V}^{\mathbb{T}}$, $\mathbb{T} \in \{\mathtt{H}, \mathtt{L}\}$, denotes the fact that v is a possible value of a variable with security type $\mathbb{T}$ and $(h, l)^{\mathtt{L}} = l$. *Declassified noninterference* considers a property on private inputs which can be observed [87, 42]. Consider a predicate ϕ on $\mathbb{V}^{\mathtt{H}}$, a program P satisfies declassified noninterference if

$$\begin{gathered}\forall l \in \mathbb{V}^{\mathtt{L}}, \forall h_1, h_2 \in \mathbb{V}^{\mathtt{H}}. \\ \phi(h_1) = \phi(h_2) \;\Rightarrow\; [\![P]\!](h_1, l)^{\mathtt{L}} = [\![P]\!](h_2, l)^{\mathtt{L}}\end{gathered}$$

4.3.2 Robust Declassification

In language-based settings, active attackers are known for their ability to control, i.e. observe and modify, part of the information used by the program. Security levels form a lattice which ordering specifies the relation between different security levels. Each program variable has two security types that model, respectively, the confidentiality level and the integrity level. In our context, all variables have only two security levels; `L` stands for *low, public, modifiable* and `H` stands for *high, private, unmodifiable*. Moreover, we assume, for each variable x, the existence of two functions, $\mathcal{C}(x)$ (confidentiality level) which shows whether the variable x is observable or not and $\mathcal{I}(x)$ (integrity level) which shows whether the variable x is modifiable or not. Henceforth each variable can have four possible security types, i.e. `LL`, `LH`, `HL`, `HH`. For example, if the variable x has type `LL` then x can be both *observed* and *modified* by the attacker, if the variable x has type `HL` then x can be *modified* by the attacker, but it cannot be *observed*, and so on.

The programs are written according to the syntax of a simple *while* language. In order to allow semantic transformations during the computation, we consider the *hole* construct, denoted by $[\bullet]$, which models the program locations where an attacker can insert code [176].

$$\begin{gathered}c ::= \mathtt{skip} \mid x := e \mid c_1; c_2 \mid \mathtt{if}\ e\ \mathtt{then}\ c_1\ \mathtt{else}\ c_2 \mid \\ \mathtt{while}\ e\ \mathtt{do}\ c \mid [\bullet]\end{gathered}$$

where $e ::= v \in \mathbb{V} \mid x \mid e_1 \text{ op } e_2$. The low integrity code, which can be inserted in holes, models the untrusted code assumed under the control of the attacker. Let $P[\vec{\bullet}]$ denote a program with holes and $\vec{a}$ (a vector of fixed attacks for each program hole) an attack, then $P[\vec{a}]$ denotes the program under the control of that attack. A *fair attack* is a program respecting the following syntax [176]:

$$a ::= \texttt{skip} \mid x := e \mid a_1; a_2 \mid \texttt{if } e \texttt{ then } a_1 \texttt{ else } a_2 \mid \texttt{while } e \texttt{ do } a$$

where all variables in e and x have security type $\texttt{LL}$. This means that fair attacks can only use program variables that are both observable and modifiable.

An important notion when dealing with active attackers is *robustness* [227]. Informally, a program is said to be *robust* when no active attacker, who controls the code in the holes, can disclose more information about the private inputs than what can be disclosed by a passive attacker, who merely observes the program's I/O. Note that, by using this attacker definition, it is possible to translate robustness into a language-based setting. Indeed, a program satisfies robust declassification if for all attacks $\vec{a}$, whenever the program $P[\vec{a}]$ can not distinguish the program behavior on the low memory, no other attack code $\vec{a}'$ can distinguish the program behavior on the low memory [176]. We formally recall the notion of robustness, for terminating programs, in presence of fair attacks [176].

$$\forall h_1, h_2 \in \mathbb{V}^{\mathtt{H}}, \forall l \in \mathbb{V}^{\mathtt{L}}, \forall \vec{a}, \vec{a}' \text{ active fair attack :}$$
$$[\![P[\vec{a}]]\!](h_1, l)^{\mathtt{L}} = [\![P[\vec{a}]]\!](h_2, l)^{\mathtt{L}} \Rightarrow [\![P[\vec{a}']]\!](h_1, l)^{\mathtt{L}} = [\![P[\vec{a}']]\!](h_2, l)^{\mathtt{L}}$$

Namely, a program is robust if any active (fair) attacker can disclose at most the same information (property of private inputs) as a passive attacker can disclose. Since a passive attacker can only observe public variables, a passive attacker is the same as the active attacker where $\vec{a} = \vec{\texttt{skip}}$.

4.3.3 Weakest Liberal Precondition Semantics

In this section we briefly present the *weakest liberal precondition* semantics, which constitutes our basic instrument for performing static analysis. Given a program c and a predicate Φ, $Wlp(c, \Phi)$ corresponds to the greatest set of input states σ such that if (c, σ) terminates in a final state σ', then σ' satisfies the predicate Φ [132, 126]. In our case, these predicates are quantifier-free first order formulas which are transformed by the *Wlp* semantics. Below, we present the rules of the semantics.

- $Wlp(\texttt{skip}, \Phi) = \Phi$
- $Wlp(x := e, \Phi) = \Phi[e/x]$
- $Wlp(c_1; c_2, \Phi) = Wlp(c_1, Wlp(c_2, \Phi))$
- $Wlp(\texttt{if } e \texttt{ then } c_1 \texttt{ else } c_2, \Phi) = (e \wedge Wlp(c_1, \Phi)) \vee (\neg e \wedge Wlp(c_2, \Phi))$

- $Wlp(\texttt{while}\ e\ \texttt{do}\ c, \Phi) = \bigvee_{i=0}^{n} Wlp_i(\texttt{while}\ e\ \texttt{do}\ c, \Phi)$
  where given $(C \stackrel{\text{def}}{=} \texttt{while}\ B\ \texttt{do}\ C_1)$

$$\begin{cases} Wlp_0(C, \Phi) \stackrel{\text{def}}{=} \neg B \wedge \Phi \\ Wlp_{i+1}(C, \Phi) \stackrel{\text{def}}{=} Wlp(\texttt{if}\ B\ \texttt{then}\ C_1\ \texttt{else skip}, Wlp_i(C, \Phi)) \end{cases}$$

Most of the above rules are easy to read. For instance, the *Wlp* of the conditional statement, given a postcondition Φ, is the disjunction of conjunctions of *Wlp* of each branch and the boolean condition of the guard. The *Wlp* of the loop statement may require the computation of some invariant condition. There exist several techniques to compute program invariants [154], but in this paper we don't consider them. The automatic generation of such invariants is is an interesting future direction we plan to explore more in detail. Unlike the weakest precondition semantics, *Wlp* determines a partial verification condition, meaning that only if the program does terminate the post-condition Φ is required to hold. In any case, for the purposes of this paper, we only consider in terminating programs, so we can establish the weakest liberal precondition in a finite number of iterations.

4.3.4 Certifying Declassification

In this section, we introduce a technique recently proposed for certifying declassification policies [42, 162] in presence of passive attackers. The method performs a backward analysis, computing the weakest precondition semantics starting from an output observation, in order to derive the maximal information that an attacker can disclose from that output observation. We use abstract interpretation for modeling the declassified properties.

Certifying declassification. In [42, 162] the authors present a method to compute the maximal private input information disclosed by a passive attacker. They consider only terminating computations, which means that the logical language does not have expressiveness limits [223]. The method has two main characteristics: it is a static analysis, and it performs a backward analysis from the observed outputs towards the inputs to protect. The first aspect is important since we would like to certify programs without executing them, the latter is important because noninterference aims at protecting the program's private input, while the attacker can only observe the public output. Both these characteristics are embedded in the weakest liberal precondition semantics of the program. Starting from a given output observation the *Wlp* semantics computes, by definition, the *greatest* set of input states leading to that observation. This allows us to calculate the private input information released by observing the public variables in output. This information corresponds exactly to the maximal private information disclosed by the program semantics. In this way, we are statically simulating the kind of analysis an attacker can perform in order to obtain the initial values of (or initial relations among) private variables. The revealed information can be modeled by a first order

predicate; the set of program states computed by the *Wlp* semantics is the one that satisfies this predicate. To be as general as possible, we consider the public observations parametric on symbolic values which are represented by logical variables. We denote as $\vec{l} = \vec{n}$ the parametric value of each low confidentiality program variable. For instance, the formula $(l = n)$ means that the program variable l has symbolic value n. For a given program, the public output observation can be expressed as a first order formula that is the conjunction of all low confidentiality variables, i.e. variables with security types `LL` or `LH`.

$$\Phi_0 \stackrel{\text{def}}{=} \{l_1 = n_1 \wedge l_2 = n_2 \wedge \cdots \wedge l_k = n_k\} = \bigwedge_{i=1}^{k} (l_i = n_i)$$

where $\forall l_i.\, \mathcal{C}(l_i) = \texttt{L}$. Without loss of generality, we assume this formula to be in a disjunctive normal form, namely a disjunction of conjunctions. We define the set of *free variables* of a logical formula Φ, denoted as $\mathcal{FV}(\Phi)$, as the set of free program variables occurring in Φ. Moreover, we assume the formula is in a normal form, where all redundancies and subsumed subformulas are removed by some simplification routine. For instance, let $(l > 1 \wedge l > 0)$ be a logical formula. We can simply write $(l > 1)$ since it subsumes the fact that $l > 0$. From now on we assume all logical formulas are in this normal form. For instance, consider the program P with $h_1, h_2 : \texttt{HH}$ and $l : \texttt{LL}$.

$$P \stackrel{\text{def}}{=} \texttt{if } (h_1 = h_2) \texttt{ then } l := 0; \texttt{ else } l := 1;$$

Then, $\mathit{Wlp}(P, l = n) = \{(h_1 = h_2 \wedge n = 0) \vee (h_1 \neq h_2 \wedge n = 1)\}$ after some simplification. If we observe $l = 0$ as the public output, all we can deduce about private inputs h_1, h_2 is that $h_1 = h_2$. Otherwise, if we observe $l = 1$, we can conclude that $h_1 \neq h_2$.

In [42, 162] this technique is formally justified by considering an abstract domain completeness-based [89] model of declassified noninterference. Here we avoid the formal details, and we simply show where and how we use abstract interpretation. Note that, usually the *Wlp* semantics is applied to specific output states in order to derive the greatest set of input states leading to those output states. The analysis starts from the state $\vec{l} = \vec{n}$, which is indeed an *abstract state*, namely a state where the private variables can have any value, while the public variables $\vec{l}$ have a specific symbolic value $\vec{n}$. This corresponds to the abstraction $\mathcal{H} \in \mathit{uco}(\wp(\mathbb{V}))$ [42] modeling the fact that the attacker is unable to observe the private data. Formally, it associates with a generic output state $\langle h, l\rangle$ the abstract state $\langle \mathbb{V}^{\texttt{H}}, l\rangle = \left\{ \langle h', l\rangle \;\middle|\; h' \in \mathbb{V}^{\texttt{H}} \right\}$. As far as the input is concerned, an abstract property is described as the set of all concrete values satisfying that property. Hence, since the *Wlp* semantics characterises the set of input states, and in particular of private inputs, then this set can be uniquely modeled as an abstract domain, i.e. the abstract property released by the program. Consider, for instance, the trivial program fragment P as above. According to the observed output value, which is either $l = 0$ or $l = 1$, we can discriminate the set of input states $\{\langle h_1, h_2, l\rangle \mid h_1 = h_2\}$

and $\{\langle h_1, h_2, l\rangle \mid h_1 \neq h_2\}$. This characterisation can be uniquely modeled by the abstract domain[1]

$$\phi = \{\top, \{\langle h_1, h_2, l\rangle \mid h_1 = h_2\}, \{\langle h_1, h_2, l\rangle \mid h_1 \neq h_2\}, \varnothing\}$$

Hence, if we declassify ϕ, the program is secure since the private information released by the program is the same as the private information that is declassified. While if, as in standard noninterference, nothing is declassified, which is modeled by the declassification policy $\phi' = \lambda x.\ \top$[2], then $\phi \sqsubseteq \phi'$, namely the policy is violated since the information released by the program is more (concrete) than what is allowed by the declassification policy.

4.3.5 Decentralized Label Model and Decentralized Robustness

The decentralized label model (DLM) was proposed as a fine-grained model to enforce end-to-end security policies for systems with mutual distrust and decentralized authorities that want to share data with each other [175]. Basically, every agent in the system defines and controls his own security policy and states which data, under his ownership, is visible (declassified) to the other agents in the system. The system itself must ensure that security policies are not circumvented and that they satisfy security requirements of all agents. More precisely, DLM consists of two basic ingredients: the *principals*, which security policies should be enforced in the model and the *labels*, which constitute the mean to enforce the security policies. Principals can be users, processes, groups, roles possibly related by an *acts-for* relation to allow the delegation of the authorities between principals. For instance, if a principal P *acts-for* a principal Q, formally $P \succeq Q$, it means that P has all privileges of Q. Labels are data annotations that express the security policy the owner sets on his data. In particular, if some data are annotated by the label *owner: reader*, the policy on that data defines the owner and the set of principals that can read the data. Security labels form a security lattice where the higher an element is in the lattice, the more restrictive are the security requirements of the data it labels. Moreover, the decentralized label model supports a declassification mechanism and allows to express policies regarding both confidentiality and integrity. The model is used to perform static analysis based on security type systems to enforce information flow policies.

Decentralized robustness is an approach proposed to enforce the robustness condition in the DLM model [74]. In particular, the fact that principals do not trust each other means that each principal can be a potential attacker. Therefore, robustness is analyzed relatively to a pair of principals: one fixes the point of view of the analysis, the other is the potential attacker. In particular, the former *fixes* which data he believes the latter can read and/or write. More formally, decentralized

[1] The elements $\top$ and $\varnothing$ are necessary for obtaining an abstract domain.

[2] Since $\forall x, y$ we have $\phi'(x) = \phi'(y)$, declassified noninterference with ϕ' corresponds to standard noninterference.

robustness is defined wrt. a pair of principals p and q, with power $\langle R_{p\to q}, W_{p\leftarrow q}\rangle$, where $R_{p\to q}$ allows to characterise the data p believes that q can read, while $W_{p\leftarrow q}$ allows to characterise the data p believes that q can modify. A system is robust wrt. all the attackers if it is robust with respect to all the pairs p, q of principals. In [74], the authors use security type systems to enforce robustness against all attackers in a simple while language with holes and explicit declassification. Basically, the type system allows the holes to be placed in low confidentiality contexts and it prevents attackers to influence the (explicit) declassification decision and the data to be declassified [176]. Once the attacker's point of view is fixed, a safe hole insertion rule defines the admissible holes for the attacker together with the variables he can modify and/or observe of the program.

4.4 Maximal Release by Active Attackers

The notion of robustness, as defined in Sect. 4.3, implicitly concerns the confidential information released by the program. Therefore, if we are able to measure the maximal release (the most concrete private property) in presence of active attacks, then we can compare it with the private information disclosed by a passive attacker and conclude about program robustness. In this section we compute (when possible) the maximal private information disclosed by an active attacker.

The active attack model we use here is more powerful than the one defined in Sect. 4.3.2, i.e. the fair attack. The attacker is now allowed to manipulate variables of security type `HL`, i.e. variables that the attacker cannot observe but can use. Indeed, `HL` is the type of those variables whose name is *visible*, i.e. usable by the attacker in his code, but whose value is not observable. In the following the active attacks are programs (without holes) such that, for all variables x occurring in the attacker's code, $\mathcal{I}(x) = L$. We call them *unfair attacks*. Unfair attacks are more general than fair attacks because they can modify variables of security type `LL` and `HL`. As an example, consider a user that wants to change his password. He can access a variable (the password) he can write but can not read (blind writing), i.e. of type `HL`. Now we want to compute the maximal information released in presence of unfair attacks.

4.4.1 Observing Input-Output

It is clear that, in order to certify the security degree of a program, also in presence of active attacks, it is important to compute what is the maximal private information released by the program. Such information can also help the programmer to understand what happens in the worst case, namely when an active attacker inserts the most harmful unfair code. Moreover, if we compute the most concrete property of private input data released by program semantics for all active attacks, we can compare it with the private information disclosed by a passive attacker and then conclude about program robustness. In this section, we consider the denotational semantics, namely input/output semantics. Hence, the program points where the

attacker can observe the low confidentiality data are the public program inputs and the public program outputs. Note that, since the active attacker can insert code (fair or unfair) in the holes, he can change the program semantics and, consequently, the property of the confidential information released by the program can be different in presence of different active attacks. Moreover, the number of possible unfair attacks may be infinite, thus, it becomes hard to compute the private information disclosed by all of them. It turns out to be impossible to characterise the maximal information released to attackers that modify the program semantics, because different attacks obtain different private properties which may be incomparable if there are infinitely many such attacks.

This problem is overcome when we consider a finite number of attackers, for instance a finite class of attacks for which we want to certify our program. In this case, we can compute the maximal information disclosed by each attack and, afterwards, we can consider the *greatest lower bound* (in the lattice of abstract domains) which characterizes the maximal information released to the fixed class of attacks. We introduce an example to illustrate this problem.

Example 4.4.1 *Consider the program* $P ::= l := h; [\bullet]$; *with variables* $h : \texttt{HH}$, $l : \texttt{LL}$ *and* $k : \texttt{HL}$. *Then the following attacks are possible:*

- $Wlp\big(l := h; [skip], \{l = n\}\big) = \{h = n\}$
- $Wlp\big(l := h; [l := k], \{l = n\}\big) = \{k = n\}$
- $Wlp\big(l := h; [l := l + k], \{l = n\}\big) = \{h + k = n\}$

For all cases the attacker discloses different information about confidential data. In particular, in the first case the attacker obtains the exact value of variable h*, in the second he obtains the exact value of variable* k *and in the third case he obtains a relation (the sum) between* h *and* k*. Note that if all active attacks were only those considered above, we could compute the greatest lower bound (*glb *for short) of private information disclosed by all of them. In this example the* glb *corresponds to the identity value of confidential variables* h *and* k*.*

However, as shown in the previous example, we can compute the private information disclosed by an attacker who fixes his attack and then check if that particular attack compromises the program robustness. To this end, we just have to use the method introduced in [42] and verify that the method described in Sect. 4.3.4 holds for the transformed program.

The previous example shows that, even for a finite number of attackers, we have to perform one *Wlp* analysis for each attack. In the following, we propose a method which performs only one analysis to deal with a (possibly infinite) set of active attacks. We follow the idea proposed in [42], where, in order to avoid an analysis for each possible output observation, the authors perform the analysis parametrically on the symbolic output observation $l = n$. In particular, an attack, being an imperative program, is just a function manipulating low integrity variables,

i.e., LL and HL variables. Hence, we propose a *Wlp* computation parametric on the expressions $f(\vec{l})$ which can be assigned by the active attacker to the low integrity variables $\vec{l}$, which we call *an attack schema* (in line with a program schema [96]). In other words, the attacker can assign to all low integrity variables an expression which can possibly depend on all the other low integrity variables. For instance, given a program where the low integrity variables are l and k, all possible unfair attacks can only use the variables l and k, namely $l := f(l,k)$ and $k = g(l,k)$, where f, g are expressions that can contain the variables l, k free.

The confidential information released by the parametric computations can be useful both the programmer and the attacker. Indeed, looking at the resulting formula which may contain f as parameter, the former can detect vulnerabilities about the confidential information released by the program, while the latter can exploit such vulnerabilities to build the most harmful attack and disclose as much as possible of the private input data. We introduce an example that shows the above method.

Example 4.4.2 *Consider the program in Ex. 4.4.1. The only low integrity variables are* $l : \texttt{LL}$ *and* $k : \texttt{HL}$*. According to the method described above we have to replace all possible unfair attacks in* $[\bullet]$ *with the attack schema* $\langle l, k\rangle := \langle f(l,k), g(l,k)\rangle$*. The initial formula is* $\Phi_0 = \{l = n\}$ *because* l *is the only program variable s.t.* $\mathcal{C}(l) = \texttt{L}$*. Thus, the Wlp calculation yields the following formula:*

$$\begin{gathered}\{f(h,k) = n\}\\ l := h;\\ \{f(l,k) = n\}\\ [\langle l, k\rangle := \langle f(l,k), g(l,k)\rangle;]\\ \{l = n\}\end{gathered}$$

The final formula $(f(h,k) = n)$ *contains information about high the confidentiality variables* h *and* k*. Thus, by fixing the unfair attacks as we did in the previous example, we can obtain information about symbolic values of* h, k *or any relation between them.*

It is worth pointing out that attack schemes capture fairly well the idea of classes of attacks which have a *similar* semantic effect (up to stuttering) on confidential information disclosed by an active attacker. We conjecture a close relationship between attack schemes and program schemes [96] and postpone their investigation as part of our future work.

4.4.2 Observing Program Traces

So far we have computed the maximal private information disclosed by an active attacker which tampers with low integrity data in predefined program points (holes) and observes the public input and the public output of the target program. In particular, the attacker can not observe the low confidentiality data in intermediate

program points, e.g. program traces. This condition is unrealistic in scenarios where the attacker can control a compromised machine. Indeed, nothing prevents him to analyze the low confidentiality data at the points he inserts low integrity code and thus to reveal private information before the overall computation terminates. This man-in-the-middle kind of attack requires to extend the analysis and consider intermediate program points as possible channels of information leakage. In many practical applications, it is common to have scenarios where a bunch of threads are running concurrently together with a malicious thread which reads the content of shared variables and dumps them in output each time that thread is scheduled to run.

In [162] the authors point out that the semantic model constitutes an important dimension for program security, the *where* dimension [202], which influences both the observation policy and the declassification policy. It seems obvious that an attacker who observes low confidentiality variables in intermediate program points can disclose more information than an attacker that observes only I/O behavior of the program. In this section, we aim to characterise the maximal information released by a program in presence of unfair attacks. In general, we can fix the set of program points where the attacker can observe the low confidentiality variables (say $\mathbb{O}$) and we can denote by $\mathbb{H}$ the set of program points where there is a hole, namely where the attacker can insert the malicious code. Moreover, we assume that the attacker can observe the low confidentiality variables for all program points in $\mathbb{H}$, namely $\mathbb{H} \subseteq \mathbb{O}$. In order to compute the maximal release of confidential information, an attacker can combine, at each observation point, the public information he can observe at that point together with the information he can derive by computing *Wlp* from the output to that observation point [162]. For instance, with the trace semantics, an attacker can observe the low confidentiality data at all intermediate program point. We first introduce an example that presents this technique for passive attackers.

Example 4.4.3 *Consider the program P with variables $l_1, l_2 : \texttt{LL}$ and $h_1, h_2 : \texttt{HH}$.*

$$P ::= \left[\begin{array}{l} h_1 := h_2; h_2 := h_2 \bmod 2; \\ l_1 := h_2; h_2 := h_1; l_2 := h_2; l_2 := l_1; \end{array} \right.$$

We want to compute the private information disclosed by an attacker that observes the program traces. As for standard noninterference, the goal is to protect the private inputs h_1 and h_2. In order to make only one iteration on the program, even when dealing with traces, the idea is to combine the Wlp *semantics computed at each observable point of execution, together with the observation of public data made at the particular observation point. We denote in square brackets the value observed at the program point under consideration. The* Wlp *calculation yields the following result.*

$$\{h_2 \bmod 2 = m \wedge h_2 = n \wedge l_2 = p \wedge l_1 = q\}$$
$$h_1 := h_2;$$
$$\{h_2 \bmod 2 = m \wedge h_1 = n \wedge l_2 = p \wedge l_1 = q\}$$
$$h_2 := h_2 \bmod 2;$$
$$\{h_2 = m \wedge h_1 = n \wedge l_2 = p \wedge [l_1 = q]\}$$
$$l_1 := h_2;$$
$$\{l_1 = m \wedge h_1 = n \wedge l_2 = p\}$$
$$h_2 := h_1;$$
$$\{l_1 = m \wedge h_2 = n \wedge [l_2 = p]\}$$
$$l_2 := h_2;$$
$$\{l_1 = m \wedge [l_2 = n]\}$$
$$l_2 := l_1;$$
$$\{l_1 = l_2 = m\}$$

For instance the information disclosed by the assignment $l_2 := l_1$ is the combination of Wlp calculation ($l_1 = m$) and attacker's observation at the same point ($[l_2 = n]$). The attacker is able to deduce the exact value of h_2. The example shows that this attacker is more powerful than the one who observes the input-output behavior; in fact, the latter can only distinguish the parity of variable h_2. This is made clear by the fact that the value of h_2's parity (m) is the value derived by the output, while the value of h_2 (n) is the value observed during *the computation.*

We recall that our goal is to compute the maximal private information release in presence of unfair attacks. The problem is similar to the one described in the previous section. Unfair attacks, by definition, manipulate (modify and use) both variables of type `LL` and `HL`. Although the attacker can observe the low confidentiality variables in presence of program holes, still he can not observe the variables of type `HL`. As a result, different unfair attacks can cause different information to be released, as it happens with attackers observing only the I/O behavior, and in general there can be an infinite number of these attacks. However, if we fix the unfair attack we can use the method described above and compute the maximal release for that particular attack.

Things change when we consider only fair attacks, i.e. manipulating only `LL` variables. The following proposition shows that we can capture all possible fair attacks with constant assignments $\vec{l} := \vec{c}$ to variables of type `LL`.

Proposition 4.4.1 *Let $P[\vec{\bullet}]$ be a program with holes and $\mathbb{H} \subseteq \mathbb{O}$. Then, all fair attacks can be written as $\vec{l} := \vec{n}$, where l : `LL`.*

Proof. In general, all fair attacks have the shape $\vec{l} := f(\vec{l})$. Moreover, $\mathbb{H} \subseteq \mathbb{O}$ hence the attacker can observe at least the program points where there is a hole. Thus, all the formal parameters of expression $f(\vec{l})$ are known. As a result, we conclude that $\vec{l} := \vec{n}$. □

Now we are able to measure the maximal private information disclosed by an active attacker. Indeed, we can use the approach of Ex. 4.4.3 and whenever we reach a program hole, we substitute it by the assignment $\vec{l} := \vec{c}$, parametric on symbolic constant values $\vec{c}$. The following example elucidates the method.

Example 4.4.4 *Consider the program P and the variables h : `HH` and l : `LL`. Let also $\mathbb{O}$ be the set of all program points.*

$$P ::= l := 0; [\bullet];\ \texttt{if}\ (h > 0)\ \texttt{then skip else}\ l := 0;$$

In presence of passive attackers, P does not release any information about the private variable h. Indeed, the output value of the public variable l is always 0. An active attacker, who can observe each program point and inject fair attacks, can disclose the following private information:

$$\begin{gathered}\{((h > 0 \wedge c = m) \vee (h \leq 0 \wedge m = 0)) \wedge c = n \wedge p = 0\}\\ l := 0;\\ \{((h > 0 \wedge c = m) \vee (h \leq 0 \wedge m = 0)) \wedge c = n \wedge [l = p]\}\\ [l := c;]\\ \{((h > 0 \wedge l = m) \vee (h \leq 0 \wedge m = 0)) \wedge [l = n]\}\\ \texttt{if}\ (h > 0)\ \texttt{then skip else}\ l := 0;\\ \{l = m\}\end{gathered}$$

Thus, the active attacker is able to disclose whether the variable h is positive or not. Hence, this is the maximal private information disclosed by an attacker who observes program traces and injects fair code in the program holes.

4.5 Enforcing Robustness

In this section we want to understand, by static program analysis, under which conditions an active attacker that transforms program semantics is unable to disclose *more* private information than a passive attacker, who merely observes the public data. The idea is to consider the *Wlp* semantics and find sufficient conditions which guarantee the program robustness. We first introduce a method to check for programs that are robust in presence of active attacks.

We know [42] that declassified noninterference is a completeness problem in abstract interpretation theory and there exist systematic methods to enforce this notion. Let $P[\vec{\bullet}]$ be a program with holes and Φ a first order formula that models the declassification policy. In order to check robustness for the program $P[\vec{\bullet}]$, we must check the corresponding completeness problem for each possible attack a, as introduced in Sect. 4.3.4, where $P[\vec{a}]$ is program P under the attack $\vec{a}$. The goal is to characterise those situations where the semantic transformation induced by the active attack does not generate incompleteness. If there is at least one attack a such that the program releases more confidential information than what is released by the policy, then the program is not robust. The following example shows the

ability of active attackers to disclose more confidential information than passive attackers.

Example 4.5.1 *Consider the program P with h : HH, l : LL.*

$$P ::= l := 0; [\bullet]\ \texttt{if}\ (h > 0)\ \texttt{then}\ (l := 1)\ \texttt{else}\ (l := l + 1);$$

Suppose the declassification policy is $\top$, i.e. nothing can be released. In presence of a passive attacker (the hole substituted by skip*) P satisfies the security policy, namely noninterference, because public output is always 1. The* Wlp *semantics formalizes this fact.*

$$\begin{gathered}\{(h > 0 \wedge n = 1) \vee (h \leq 0 \wedge n = 1)\} = \{n = 1\}\\ l := 0;\\ \{(h > 0 \wedge n = 1) \vee (h \leq 0 \wedge n = l + 1)\}\\ \texttt{if}\ (h > 0)\ \texttt{then}\ (l := 1)\ \texttt{else}\ (l := l + 1);\\ \{l = n\}\end{gathered}$$

Suppose now an active attacker inserts the code $l := 1$. In this case the Wlp *semantics shows that the attacker is able to distinguish positive values of the private variable h from non positive ones. Using the* Wlp *calculation parametric on public output $\{l = n\}$ we have the following result.*

$$\begin{gathered}\{(h > 0 \wedge n = 1) \vee (h \leq 0 \wedge n = 2)\}\\ l := 0;\\ \{(h > 0 \wedge n = 1) \vee (h \leq 0 \wedge n = 2)\}\\ [l := 1;]\\ \{(h > 0 \wedge n = 1) \vee (h \leq 0 \wedge n = l + 1)\}\end{gathered}$$

The final formula shows that the attacker is able to distinguish values of h greater than 0 from values less or equal than 0 by observing, respectively, the values 1 or 2 of the public variable l. We can conclude that program P is not robust and, as a result, the active attackers are indeed more powerful than the passive ones.

If we had a method to compute the maximal private information release in presence of unfair attacks, then we could check the program robustness by comparing it with the information disclosed by a passive attacker. Unfortunately, in the previous section, we have seen that it is not possible to compute the maximal information released for all possible attacks, as there can be infinitely many. Hence, we should look for methods to check program robustness without computing the maximal information release.

4.5.1 Robustness by *Wlp*

In this section we first distinguish between active attacks with different capabilities and, afterwards, we present and prove the correctness of our approach to certify

robust programs. The proof is organised as follows: we start with a lemma that applies to sequential programs with one hole only, then we give a a theorem that generalizes the lemma to sequential programs with more holes and conclude with another theorem that applies the robustness condition to all terminating while programs.

We first make some considerations about the logical formulas and the set of program states they represent. The free variables of the output observation formula Φ_0 correspond to the set of low confidentiality variables LL and LH, namely

$$\mathcal{FV}(\Phi_0) = \{x \in Var(\Phi_0) | \mathcal{C}(x) = \mathtt{L}\}.$$

If a low confidentiality variable does not occur free at some program point, it means that the variable was previously, wrt. backward analysis of *Wlp*, substituted by an expression that does not contain that variable. This means that, the variable can have any value in that program point. From the viewpoint of the information flow, even if the variable contains some confidential information at that point, this information is irrelevant to the analysis, because subsequently the variable is going to be overwritten and therefore the private information it contains will never be revealed by the public outputs.

Our aim is to generalise the most powerful active attacks and study their impact on program robustness. As a first try, we can represent all active attacks by a constant assignment to low integrity variables. Recall that the attacker observes only the input/output value of low confidentiality variables, i.e. LL and LH variables. Then the following example shows that this is not sufficient and there exist more powerful attacks that disclose more private information and thus break robustness.

Example 4.5.2 *Consider the program P with variables l : LL, k : LL, h : HH and a declassification policy that allows nothing to be released about the private variables.*

$$P ::= \left[\begin{array}{ll} k := h; [\bullet]; & \\ \texttt{if } (l = 0) & \texttt{then } (l := 0; k := 0) \\ & \texttt{else } (l := 1; k := 1); \end{array}\right.$$

First notice that P does not release private information in presence of a passive attacker. Indeed, the assignment of h to k is subsequently overwritten by the constants 0 or 1 and it only depends on the variation of public input l. If it was possible to represent all active attacks by the constant assignments then P would be robust. In fact, if the attacker assigns constants c_1 and c_2, respectively, to variables l and k, the Wlp calculation deems the program robust.

$$\begin{array}{c} \{(c_1 = 0 \wedge m = 0 \wedge n = 0) \vee (c_1 \neq 0 \wedge m = 1 \wedge n = 1)\} \\ k := h; \\ [l := c_1; k := c_2;] \\ \{(l = 0 \wedge m = 0 \wedge n = 0) \vee (l \neq 0 \wedge m = 1 \wedge n = 1)\} \\ \texttt{if } (l = 0) \texttt{ then } (l := 0; k := 0) \texttt{ else } (l := 1; k := 1); \\ \{l = m \wedge k = n\} \end{array}$$

The final formula shows that the program satisfies noninterference. But if we assign to the low integrity variables an expression depending on the other low integrity variables, then we obtain a more powerful attack, which makes P not robust. For instance, the assignment $a ::= l := k;$ allows the attacker to distinguish the zeroness of private variable h.

$$\begin{gathered}\{(h = 0 \wedge m = 0 \wedge n = 0) \vee (h \neq 0 \wedge m = 1 \wedge n = 1)\} \\ k := h; \\ \{(k = 0 \wedge m = 0 \wedge n = 0) \vee (k \neq 0 \wedge m = 1 \wedge n = 1)\} \\ [l := k;] \\ \{(l = 0 \wedge m = 0 \wedge n = 0) \vee (l \neq 0 \wedge m = 1 \wedge n = 1)\}\end{gathered}$$

Definitely, program P is not robust and therefore we cannot reduce active attacks to a constant assignment to the low integrity variables.

In general, an active attack is a piece of code that uses low integrity variables, namely a function on low integrity variables. If we assign a constant value to the low integrity variables then we may erase the high confidentiality information that these variables may have accumulated before reaching the program point or we can ignore the fact that another variable, which may contain private information, can be assigned to our variable, as shown in Ex. 4.5.2.

We use the ideas to introduce a *sufficient* condition which ensures program robustness. Recall that the observed public output is formally represented as a first order formula, Φ_0, which corresponds to the conjunction of program variables x such that $\mathcal{C}(x) = \mathtt{L}$, and it is parametric on the observed public output n_i, namely

$$\Phi_0 = \bigwedge_{i=1}^{k} (l_i = n_i) \text{ and } \forall i.\mathcal{C}(l_i) = \mathtt{L}$$

We first describe a sufficient condition for programs where the holes are not nested in the control structures of the program. This is obtained in two steps, the lemma shows the result for programs with only one hole, while the first theorem extends this result to programs with an arbitrary number of holes. Afterwards, we show how the approach can be extended in order to characterise robustness also for programs where the holes occur in arbitrary places in the program. We denote by $\bullet_i$ the i-th hole in P and by P_i the portion of code in P after the hole $\bullet_i$ where all the following holes ($\bullet_j$, with $j \in \mathbb{H}$, $j > i$) are replaced with **skip**. Then $\Phi_i = Wlp(P_i, \Phi_0)$ is the formula corresponding to the execution of subprogram P_i.

Lemma 4.5.1 *Let $P = P_2; [\bullet]; P_1$ be a program (P_1 without holes, possibly empty). Let $\Phi = Wlp(P_1, \Phi_0)$. Then P is robust wrt. unfair attacks if $\forall v \in \mathcal{FV}(\Phi).\mathcal{I}(v) = \mathtt{H}$.*

Proof. We prove this theorem by induction on the attacks structure and on the size of its derivation. In particular, we prove that for any attack a, $Wlp(a, \Phi) = \Phi$,

namely the formula Φ does not change, hence from the semantic point of view, the attack behaves like **skip**, namely like a passive attacker. Note that here we consider unfair attacks, hence the attacker can use both `LL` and `HL` variables.

- a ::= `skip`: The initial formula Φ does not change, namely $Wlp(\texttt{skip}, \Phi) = \Phi$, and the attacker acts as a passive one.
- a ::= l := e: By definition of active attack we have $\mathcal{I}(l) = \texttt{L}$ and by hypothesis variable l does not occur free in Φ. Applying the *Wlp* definition for assignment, we have $Wlp(l := e, \Phi) = \Phi[e/l] = \Phi$.
- $a ::= c_1; c_2$: By inductive hypothesis we have $Wlp(c_1, \Phi) = Wlp(c_2, \Phi) = \Phi$, being attacks of smaller size. The *Wlp* definition for sequential composition states that $Wlp(c_1; c_2, \Phi) = Wlp(c_1, Wlp(c_2, \Phi)) = \Phi$
- $a :=$ `if` B `then` c_1 `else` c_2: By inductive hypothesis (applied to an attack of smaller size) we have $Wlp(c_1, \Phi) = Wlp(c_2, \Phi) = \Phi$. Applying the definition of *Wlp* for the conditional construct $Wlp(\texttt{if } B \texttt{ then } c_1 \texttt{ else } c_2, \Phi) = (B \wedge Wlp(c_1, \Phi)) \vee (\neg B \wedge Wlp(c_2, \Phi)) = (B \wedge \Phi) \vee (\neg B \wedge \Phi) = \Phi$.
- $a ::=$ `while` B `do` c: By hypothesis we consider terminating programs, so the `while` loop halts in a finite number of iterations. Applying the inductive hypothesis to the command c we have $Wlp(c, \Phi) = \Phi$, so each iteration the formula does not change. Moreover, if the guard is *false* the formula remains unchanged too. Applying *Wlp* rule for the `while` loop and the inductive hypothesis we have:
  $Wlp(\texttt{while } B \texttt{ do } c, \Phi) = (\neg B \wedge \Phi) \vee (B \wedge \Phi) \vee \cdots \vee (B \wedge \Phi) \vee (B \wedge \Phi) = \Phi$

□

Theorem 4.5.1 *Let $P[\vec{\bullet}]$ be a program. Then we say that P is robust wrt. unfair attacks if* $\forall i \in \mathbb{H}. \forall v \in \mathcal{FV}(\Phi_i). \mathcal{I}(v) = \texttt{H}$.

Proof. Suppose P has n holes:

$$P \equiv P'_{n+1}; [\bullet_n]; P'_n \dots P'_2; [\bullet_1]; P'_1$$

Let us define the following programs for $1 \leq i \leq n+1$

$$P_i \stackrel{\text{def}}{=} \begin{cases} P'_1 & \text{if } i = 1 \\ P'_i P_{i-1} & \text{otherwise} \end{cases}$$

Namely P_i is the portion of code in P after the hole $\bullet_i$ where all the following holes ($\bullet_j$, with $j \in \mathbb{H}$, $j > i$) are replaced with **skip**. We prove by induction on n that $\forall 1 \leq i \leq n.\ P'_{i+1}; [\bullet_i]; P'_i; [\bullet_{i-1}]; \dots; [\bullet_1]; P'_1$ is robust wrt. unfair attacks. By proving this fact, we prove the thesis since for $i = n$ we obtain exactly P.

BASE: Consider the first hole from the end of the program P, i.e. $P'_2; [\bullet_1]; P'_1$. Then by Lemma 4.5.1 we have that $P'_2; [\bullet_1]; P'_1$ is robust, being P'_1 without holes by construction. This implies that any active attacker can disclose the same information as the passive (`skip`) attacker can do, hence $\bullet_1$ can be replaced with `skip`, namely $P'_2[\bullet_1]P'_1$ can be substituted by P_2 in P without changing the robustness property of P.

INDUCTIVE STEP: Suppose, by inductive hypothesis, $P'_i; [\bullet_{i-1}]; P'_{i-1}; \ldots; P'_2; [\bullet_1]; P'_1$ is robust. This means that, exactly as for the base of the induction, the holes are useless for the attacker, therefore we can substitute all the $\bullet_j$ with `skip` obtaining a program (from the robustness point of view) equivalent to P_i. Hence, $P'_{i+1}; [\bullet_i]; P'_i; [\bullet_{i-1}]; \ldots; [\bullet_1]; P'_1 \equiv P'_{i+1}; [\bullet_i]; P_i$, and the robustness of the last program holds by Lemma 4.5.1, being P_i without holes by construction.

In this way we prove that $P \equiv P'_{n+1}; [\bullet_n]; P_n$ is robust. □

In other words, the fact that a low integrity variable is not free in the formula implies that the information in the corresponding program point can not be exploited to reveal confidential information. In this case we can say that a generic active attacker is not stronger than a passive one. Before showing what happens for the control structures, let us introduce an example that illustrates Theorem 4.5.1.

Example 4.5.3 *We check robustness for the program P with variables l : `LL`, h : `HH` and k : HL.*

$$P ::= \left[\begin{array}{l} l := h + l; [\bullet]; l := 1; k := h; \\ \texttt{while}\ (h > 0)\ \texttt{do}\ (l := l - 1; l := h); \end{array}\right.$$

Analysing P from the hole $[\bullet]$ to the end we have:

$$\begin{array}{c} \{(h \leq 0 \wedge n = 1) \vee (h > 0 \wedge n = 0)\} \\ l := 1; k := h; \\ \{(h \leq 0 \wedge l = n) \vee (h > 0 \wedge n = 0)\} \\ \texttt{while}\ (h > 0)\ \texttt{do}\ (l := l - 1; l := h); \\ \{l = n\} \end{array}$$

The formula $\Phi = (h \leq 0 \wedge n = 1) \vee (h > 0 \wedge n = 0)$ meets the conditions of Theorem 4.5.1. We can conclude the program P is robust. Intuitively, even though the value of private input h flows to the public variable l ($l := l + h$), this relation is immediately lost when we assign the constant 1 ($l := 1$) after the hole.

The following example shows that Theorem 4.5.1 is just a sufficient condition, namely there exists a robust program that violates the conditions of the theorem. This is because Theorem 4.5.1 is a local condition wrt. robustness, however one may need to analyze the entire program to have the information about the revealed confidential information.

Example 4.5.4 *Consider the program*

$$P ::= \left[\begin{array}{l} l := h; l := 1; [\bullet]; \\ \texttt{while}\ (h = 0)\ \texttt{do}\ (h := 1; l := 0); \end{array} \right.$$

where $h : \texttt{HH}$ *and* $l : \texttt{LL}$. *The precondition of the* `while` *is:*

$$Wlp\,(\texttt{while}\ (h = 0)\ \texttt{do}\ (h := 1; l := 0), \{l = n\}) = \\ \{(h = 0 \wedge n = 0) \vee (h \neq 0 \wedge l = n)\}$$

This formula does not satisfy the conditions of Theorem 4.5.1, since it contains a free occurrence of a low integrity variable, namely $l = n$. *However, we can see that program* P *is robust. No modification of the public variable* l *contains information about the private variable* h *because the guard of the* `while` *loop depends exclusively on the private variables. Every terminating attack modifies the subformula* $\{l = a\}$ *and influences the final value of the observed public output. Moreover, the private information obtained by the assignment* $l := h$ *is erased by the successive assignment* $l := 1$. *So the only confidential information released by* P *concerns the zeroness of* h, *the same as for the passive attacker. This means that* P *is robust and Theorem 4.5.1 is a sufficient and not necessary condition for checking robustness.*

We now show how Theorem 4.5.1 applies to programs where the hole occurs in a conditional or in a loop. As the following theorem shows, in these cases we need to apply recursively Theorem 4.5.1 to the formula corresponding to each branch. It is worth noting that the loop can be unfolded a finite number of times until we reach the invariant formula (see the *Wlp* rule for `while` in Sect. 4.3.3), as the computations are terminating.

Theorem 4.5.2 *Let* $P_c[\vec{\bullet}] \equiv \texttt{if}\ B\ \texttt{then}\ P_1[\vec{\bullet}]\ \texttt{else}\ P_2[\vec{\bullet}]$ *and* $P_w[\vec{\bullet}] \equiv \texttt{while}\ B\ \texttt{do}\ P[\vec{\bullet}]$ *be a program with holes and a first order formula* Φ. *Then,*

- $P_c[\vec{\bullet}]$ *is robust wrt. unfair attacks iff* $P_1[\vec{\bullet}]$ *and* $P_2[\vec{\bullet}]$ *are robust wrt. unfair attacks and post-condition* Φ.
- $P_w[\vec{\bullet}]$ *is robust wrt. unfair attacks iff* $P[\vec{\bullet}]$ *is robust wrt. unfair attacks and post-conditions* $Wlp_i(P_w[\vec{\bullet}], \Phi)$

Proof. We do induction on the structure of $P_1[\vec{\bullet}]$; the other case is symmetric. If $P_1[\vec{\bullet}]$ straight line program with holes (as in the hypothesis of Theorem 4.5.1), we apply the theorem to check robustness. Otherwise, $P_1[\vec{\bullet}]$ is a conditional and it trivially holds from the induction hypothesis.
In the case of a loop we need to apply the recursive computation as described in Sect. 4.3.3. If $P[\vec{\bullet}]$ is a straight line program we apply Theorem 4.5.1 as before and check, at each step of the *Wlp* computation whether low integrity variables occur in the formula, when we reach the hole. Note that the occurrence of the loop guard B in the formula makes sure that the active attacker never influences the variables

of B. In this way, we are sure that if the condition is verified the formula remains unchanged for all active attacks. Otherwise, if $P[\vec{\bullet}]$ is a loop or a conditional we apply the induction hypothesis and we are done. □

The result above shows how to treat situations where the construct $[\bullet]$ may be placed in an arbitrary depth inside an `if` conditional or a `while` loop. The following example describes this situation.

Example 4.5.5 *Consider the program P*

$$P ::= \left[\begin{array}{ll} k := h \ mod\ 3; & \\ \texttt{if}\ (h \ mod\ 2 = 0) & \texttt{then}[\bullet]; l := 0; k := l; \\ & \texttt{else}\ l := 1; \end{array}\right.$$

where h : `HH`, l : `LL` and k : `LL`. Applying the weakest liberal precondition rules to the initial formula $\{l = m \wedge k = n\}$ we have:

$$\left\{\begin{array}{c} (h \ mod\ 2 = 0 \wedge m = 0 \wedge n = 0) \vee \\ (h \ mod\ 2 \neq 0 \wedge m = 1 \wedge k = n) \end{array}\right\}$$
$$\texttt{if}\ (h \ mod\ 2 = 0)\ \texttt{then}\ [\bullet]; l := 0; k := l;\ \texttt{else}\ l := 1;$$
$$\{l = m \wedge k = n\}$$

The subformula corresponding to the **then** *branch (which contains the hole $[\bullet]$) satisfies the conditions of Theorem 4.5.1, therefore P is robust. Every attack at this point manipulates the variables l, k which are immediately substituted by the constant 0 and therefore lose all the private information that have accumulated.*

Note that the theorems enforce the invariant stating that the first order formula determined by the active attack does not change, compared to the formula determined by the passive attack. In particular, Theorem 4.5.1 proves the security condition, while Theorem 4.5.2 models the fact that such condition should be applied recursively in case of conditionals and loops. In the next section we present an algorithmic approach that puts all the pieces together.

4.5.2 An Algorithmic Approach to Robustness

In this section we present our approach algorithmically in order to make clear how the above theorems apply to terminating while programs. In particular, $Robust(P[\vec{\bullet}], \Phi, \mathcal{S})$ is the main procedure that takes as input a program with holes $P[\vec{\bullet}]$, a first order formula Φ and a set of low integrity variables S and, if it returns a formula, the program is robust and this formula corresponds to the private information disclosed to both passive and active attackers, otherwise (if it returns false) we don't know whether the program is robust or not. The procedure $Check(\Phi, \mathcal{S})$ corresponds to the security condition, namely, it returns true if no low integrity variables in $\mathcal{S}$ occur in Φ as well. Moreover, we assume that we have a procedure

that transforms a first order formula in the normal form to reduce the false alarms during the analysis. The algorithm runs recursively over the syntactical structure of the while program (with holes) and, at each step it applies the rules of *Wlp* semantics, as described in Sect. 4.3.3. The procedure $Compute(\texttt{while}\ B\ \texttt{do}\ c, \Phi, \mathcal{S})$ checks whether the formula remains unchanged for the while loop. In particular, this corresponds to the unfolding of the loop, with a finite number of conditionals and then applies a finite number of times Theorem 4.5.2.

$$Robust(P[\vec{\bullet}], \Phi, \mathcal{S}) :$$

$$\left[\begin{array}{ll}
\textbf{case}(P[\vec{\bullet}]) : & \\
 & \\
[\bullet] : & Check(\Phi, \mathcal{S}) \\
\texttt{skip} : & \Phi \\
x := e : & \Phi[e/x] \\
P_1[\vec{\bullet}]; P_2[\vec{\bullet}] : & \Phi' := Robust(P_2[\vec{\bullet}], \Phi, \mathcal{S}) \\
 & Robust(P_1[\vec{\bullet}], \Phi', \mathcal{S}) \\
\texttt{if}\ B\ \texttt{then}\ P_1[\vec{\bullet}]\ \texttt{else}\ P_2[\vec{\bullet}] : & (B \wedge Robust(P_1[\vec{\bullet}], \Phi, \mathcal{S})) \vee \\
 & (\neg B \wedge Robust(P_2[\vec{\bullet}], \Phi, \mathcal{S})) \\
\texttt{while}\ B\ \texttt{do}\ P_1[\vec{\bullet}] : & Compute(\texttt{while}\ B\ \texttt{do}\ P_1[\vec{\bullet}], \Phi, \mathcal{S})
\end{array}\right.$$

$$Check(\Phi, \mathcal{S}) : \left[\begin{array}{ll}
\text{Normalize the formula } \Phi & \\
\texttt{if}\ \mathcal{FV}(\Phi) \cap \mathcal{S} = \emptyset & \text{return true} \\
\text{otherwise} & \text{return false}
\end{array}\right.$$

$$Compute(\texttt{while}\ B\ \texttt{do}\ c, \Phi, \mathcal{S}) : \left[\begin{array}{l}
\Phi_{i+1} := \neg B \wedge \Phi \\
result := \Phi_{i+1} \\
\texttt{do} \\
\quad \Phi_i := \Phi_{i+1} \\
\quad \Phi_{i+1} := Robust(\texttt{if}\ B\ \texttt{then}\ c\ \texttt{else skip}, \Phi_i, \mathcal{S}) \\
\quad result \vee := \Phi_{i+1} \\
\texttt{while}\ \Phi_i \neq \Phi_{i+1}
\end{array}\right.$$

4.5.3 Robustness on Program Traces

In this section, we want to find local conditions that imply robustness in presence of active attackers that observe the trace semantics of the program. In other words, we want to find the analogous of Theorem 4.5.1 when dealing with trace semantics. Note that, in this case, the problem becomes quite different because the attacker is still able to modify low integrity variables, however he can also *observe* low confidentiality variables at the holes. The main problem is that the attacker can assign variables of type `HL` to variables of type `LL`, observe the corresponding trace and immediately disclose the value of `HL` variables. Hence, it becomes necessary to analyse the global program behavior in order to check robustness for all possible unfair

attacks. On the other hand, if we consider fair attacks, i.e., attacks that manipulate only LL variables, the attacker's capability to observe program points where the hole occurs allows us to reduce all possible attacks to constant assignments to variables of type LL.

By using the method introduced in [162], illustrated for active attackers in Sect. 4.4.2, we are now able to state a sufficient condition of robustness in presence of fair attacks for the trace semantics. The idea is that an attacker can combine the public information he observes at a program point with the information he can deduce by computing the *Wlp* from the output to that program point. Moreover, the attacker can manipulate program semantics by inserting fair attacks in the holes. If the formula resulting from *Wlp* semantics of the subprogram after the hole does not contain free variables of type LL then we can conclude that the program is robust. The following example shows a robustness condition similar to Theorem 4.5.1.

Example 4.5.6 *Consider the program P and variables l : LH, k : LL and h_1, h_2, h_3 : HH:*

$$P ::= k := h_1 + h_2; [\bullet]; k := h_3 \bmod 2; l := h_3; l := k;$$

A passive attacker who observes all program points discloses the following private information.

$$\begin{gathered}
\{h_3 \bmod 2 = m \wedge h_3 = n \wedge l = p \wedge h_1 + h_2 = q\} \\
k := h_1 + h_2; \\
[skip;] \\
\{h_3 \bmod 2 = m \wedge h_3 = n \wedge l = p \wedge [k = q]\} \\
k := h_3 \bmod 2; \\
\{k = m \wedge h_3 = n \wedge [l = p]\} \\
l := h_3; \\
\{k = m \wedge [l = n]\} \\
l := k; \\
\{l = k = m\}
\end{gathered}$$

Hence, a passive attacker reveals the value of variable h_3 and the sum of variables h_1 and h_2. Note that no fair attack (in our case manipulating k) can do better, since the subformula corresponding to the information disclosed by the attacker does not contain the variable k : LL free. Thus, no constant assignment influences the private information released. Indeed, if we compute the information disclosed in presence of a fair attack the final formula is the same.

$$\begin{gathered}
\{h_3 \bmod 2 = m \wedge h_3 = n \wedge l = p \wedge h_1 + h_2 = r\} \\
k := h_1 + h_2; \\
\{h_3 \bmod 2 = m \wedge h_3 = n \wedge l = p \wedge q = d_1 \wedge [k = r]\} \\
[k := d_1;] \\
\{h_3 \bmod 2 = m \wedge h_3 = n \wedge l = p \wedge [k = q]\}
\end{gathered}$$

Note that, it is useless to consider the observed value of LL *variable before the hole because the attacker knows exactly what fair attack he is going to inject.*

Now we can introduce a sufficient condition to check robustness for trace semantics. Basically, the idea is to extend Theorem 4.5.1 to traces. We have first to note that in Theorem 4.5.1 we deal with unfair attackers, which can use also variables of type HL. In the trace semantics this may be a problem whenever attackers can observe low confidentiality data in at least one point where they can inject their code, i.e. if $\mathbb{H} \cap \mathbb{O} \neq \varnothing$. In particular what may happen is that the attacker can use variables of type HL and observe the result immediately after, possibly disclosing the value of these variables. This clearly means that the program is not robust as the following example shows.

Example 4.5.7 *Consider the program*

$$P := l := k \bmod 2; [\bullet];\ \texttt{if}\ (h = 0)\ \texttt{then}\ l := 0\ \texttt{else}\ l := 1;$$

where l : LL, k : HL *and* h : HH. *We want to check robustness in presence of unfair attackers who observe each program point. A passive attacker discloses the zeroness of variable* h *and the parity of variable* k. *The information released at the hole is then computed as follows.*

$$\begin{gathered}\{(h = 0 \wedge n = 0) \vee (h \neq 0 \wedge n = 1)\} \\ \texttt{if}\ (h = 0)\ \texttt{then}\ l := 0\ \texttt{else}\ l := 1; \\ \{l = n\}\end{gathered}$$

This formula satisfies the conditions of Prop. 4.5.1: no low integrity variable occurs free in the formula. However, if we use the unfair attack (e.g., $l := k$*), we can see that the program releases the exact value of the private variable* k.

$$\begin{gathered}\left\{\begin{array}{c}((h = 0 \wedge n = 0) \vee (h \neq 0 \wedge n = 1)) \wedge \\ k = p \wedge k \bmod 2 = q\end{array}\right\} \\ l := k \bmod 2; \\ \{((h = 0 \wedge n = 0) \vee (h \neq 0 \wedge n = 1)) \wedge k = p \wedge [l = q]\} \\ [l := k;] \\ \{((h = 0 \wedge n = 0) \vee (h \neq 0 \wedge n = 1)) \wedge [l = p]\}\end{gathered}$$

We conclude that program P *is not robust (wrt. unfair attacks) even though the conditions of Theorem 4.5.1 are satisfied.*

The following proposition determines a sufficient condition to check robustness for program traces, depending on the relation between the hole points $\mathbb{H}$ and observable points $\mathbb{O}$. In particular, if the attackers observe low confidentiality data in at least one hole point, i.e. $\mathbb{H} \cap \mathbb{O} \neq \varnothing$, then we can check robustness only wrt. the fair attacks, otherwise we can consider general unfair attacks. In fact, when $\mathbb{H} \cap \mathbb{O} = \varnothing$ the attacker cannot combine his capabilities of observing low confidentiality variables and of modifying low integrity variables, and thus can not violate robustness.

Proposition 4.5.1 *Consider $P[\bullet]$ and $\Phi_i = Wlp(P_i, \Phi_0)$ (where P_i is obtained as in Theorem 4.5.1). Then we have that:*

1. *If $\mathbb{H} \cap \mathbb{O} \neq \varnothing$ then P is robust wrt. fair attacks if $\forall i \in \mathbb{H}. \forall v \in \mathcal{FV}(\Phi_i). \mathcal{I}(v) =$ H.*
2. *If $\mathbb{H} \cap \mathbb{O} = \varnothing$ then P is robust wrt.* unfair *attacks if $\forall i \in \mathbb{H}. \forall v \in \mathcal{FV}(\Phi_i). \mathcal{I}(v) =$ H.*

Proof. Consider the program P. First note that the difference between observing I/O semantics and trace semantics consists in the fact that the attacker make more observations during the computation. Hence, we can define an enriched weakest precondition function: $Wlp'(c, \phi) \stackrel{\text{def}}{=} Wlp(c, \phi \wedge \phi')$, where $\phi' = true$ if the corresponding program point is not in $\mathbb{O}$, ϕ' is the observable property otherwise. At this point, by using Wlp' instead of Wlp we can apply Theorem 4.5.1 with the following restrictions:

1. If $\mathbb{H} \cap \mathbb{O} \neq \varnothing$ then the attacker can use variables of type HL and observe the result at the same time, disclosing the HL variables and violating robustness. In particular, if the program has l : LL and k : HL, then the attacker can always insert the code $l := k$, and by observing the result can directly know the value of k violating confidentiality and, obviously, robustness. This is not a problem for fair attackers, since these attackers can not use variables of type HL.
2. If $\mathbb{H} \cap \mathbb{O} = \varnothing$ then the unfair attacker can not observe the result of the added code and therefore robustness can again hold, at least when the sufficient condition of Theorem 4.5.1 is satisfied.

□

4.5.4 *Wlp* vs Security Type System

In [176] the authors define the notion of robustness in presence of active attackers and enforce it by using a security type system. The active attacker can replace the holes by fair attacks. The key result of the article states that typable programs satisfy robust declassification. Thus, a program satisfies robustness whenever the holes are not placed into high confidentiality contexts. This is achieved by introducing a security environment and a program counter *pc* which traces the security context and avoids implicit flows. The following typing rule captures the cases where the construct $[\bullet]$ is admissible.

$$\frac{\mathcal{C}(\mathrm{pc}) \in L_C}{\Gamma, \mathrm{pc} \vdash \bullet}$$

Let A be the attacker's code, then $L_C \stackrel{\text{def}}{=} \{l | \mathcal{C}(l) \sqsubseteq \mathcal{C}(A)\}$, namely L_C is the set of variables whose confidentiality level is not greater than attacker's confidentiality

level. Hence, an active attacker that manipulates these variables does not obtain more confidential information. The type system is imprecise with respect to standard noninterference since it rules out all programs containing low assignments under high guards or any sub-command with an explicit assignment from a high variable to a low variable. Basically, the type system accepts programs that *associate* low with low, high with high and do not use high expressions on guards of conditionals or loops. This corresponds to a trace-based characterization of noninterference where the attacker can observe the low variables at each program point. If we ignore the explicit declassification construct (*declassify(e)*) and consider only programs with holes, the typing rule requires the hole to occur in a low confidentiality security context. Namely the program is robust if there is no interaction between high and low, neither explicit nor implicit. For explicit declassification, the rule requires that the declassification occurs in a low confidentiality and a high integrity program context, namely the guard of a conditional or a loop is allowed to use only variables of security type `LH`. Moreover, only high integrity variables can be declassified, i.e. only declassification from variables of security type `HH` to variables of security type `LH` is allowed. Putting all together, the type system approach would accept programs that never branch on a secret value (unless each branch assigns only to high) and admit explicit flow (from high to low) in certain program points because of declassification.

Our approach, in particular Theorem 4.5.2, captures exactly those situations where the hole occurs in some confidentiality context (possibly high) and, nevertheless, the fair attack does not succeed, namely where there are no low integrity variables in the corresponding first order formula. If our condition holds, we are more precise to capture the main goal of robustness, i.e. that an active attacker does not disclose more private information than a passive one, as we perform a flow sensitive analysis. Indeed, if the target program has some intended global interference (the *what* dimension in [202]), the type system is unable to capture it (as it only considers the *where* dimension in [202]), while our approach characterizes robustness with respect to a program and a global declassification policy. Moreover, our method deals with more powerful active attacks, the unfair attacks, which can manipulate code that contains variables with security type `LL` and `HL`. However, both these approaches study program robustness as a local condition and therefore can not provide a complete characterisation of robustness: Theorem 4.5.2 provides only a sufficient condition and the type system is not complete. We can say that, when it can be applied, our semantic-based method is more precise, in the sense that it generates less *false alarms*, than the type-based one. As an example, consider the program $P ::= [\bullet]; \texttt{if } h > 0 \texttt{ then } l := 0 \texttt{ else } l := 0$ where $h : \texttt{HH}$ and $l : \texttt{LL}$. Our method certifies this program as robust since, there are no low integrity variables in the formula corresponding to the *Wlp* semantics of the `if` statement. If we try to type check this program using the rules in [176] we notice that the environment before hole is of high confidentiality. Thus, this program is deemed not robust.

We have to note that our approach, if compared with the type-based one, loses

effectiveness in order to keep precision, i.e. in order to reduce false alarms. Indeed, in the future, in order to make our certification approach systematic, we may have to weaken the semantic precision.

4.6 Relative Robustness

So far, we have provided only sufficient conditions to enforce robust programs. The problem is that an active attacker transforms the program semantics and these transformations can be infinitely many or of infinitely many kinds. This may be an issue, first of all because it becomes hard to compute the private information released by all the active attacks (as underlined in Sect. 4.4), but also because, in some restricted contexts, robustness can be too strong a requirement.

In this section, we consider a restricted class of active attacks and check robustness wrt. these attacks. Then, the goal is to check whether the program releases more private information than a passive attacker, for a class of active attacks. Thus, we define a relaxed notion of robustness, called *relative robustness*.

Definition 4.6.1 *Let $P[\vec{\bullet}]$ be a program and $\mathcal{A}$ a set of attacks. The program is relatively robust iff for all $\vec{a} \in \mathcal{A}$, then $P[\vec{a}]$ does not release more confidential information than $P[\overrightarrow{skip}]$.*

Recall that we model the information disclosed by the attacker by first order formulas, which we interpret by means of the abstract domains in the lattice of abstract interpretations, as explained in Sect. 4.3.4. In particular, if an attacker a_1 discloses more private information than an attacker a_2, it means that the abstract domain modeling the private property revealed by a_1 is contained in the abstract domain modeling the private property revealed by a_2.

In order to check relative robustness we can start by computing the confidential information disclosed by all attacks, then calculate the greatest lower bound of this information and finally compare it with the confidential information disclosed by a passive attacker. As a result, given a program and a set of attacks we can statically certify the security degree of that program wrt. a finite class of attacks. A programmer who wants to certify program robustness in presence of a fixed class of attacks, will have to declassify at least the *glb* of the private information disclosed by all these attacks.

Consider Ex. 4.4.1. We pointed out that different active attacks can disclose different properties of the private information, for this reason the program P was not robust. Now, consider a restriction of the possible active attacks, for example, consider the fair attacks only. This implies that the attacker can use only the variable l and thus derive information only about the private variable h. But the program P already releases through l the exact value of h and consequently no attack involving variable l can disclose more private information. Thus, we conclude that program P satisfies relative robustness wrt. the class of fair attacks.

We can now extend Theorem 4.5.1 to cope with relative robustness. Recall that this theorem provides a sufficient condition for robustness requiring that the formulas before each hole do not contain any low integrity variable. We weaken this sufficient condition by requiring that the formulas before each hole do not contain any of the variables modifiable and usable by the attacks in $\mathcal{A}$. In particular, both Prop. 4.6.1 and Prop. 4.6.2 can be easily extended to programs with more holes occurring at different depths, exactly the same way as we did when proving Theorem 4.5.1 from Lemma 4.5.1 and Theorem 4.5.2 from Theorem 4.5.1. The next proposition is a rewriting of Lemma 4.5.1 for relative robustness.

Proposition 4.6.1 *Let $P = P_2; [\bullet]; P_1$ be a program (where P_1 is without holes). Let $\Phi = Wlp(P_1, \Phi_0)$. P is relatively robust wrt. unfair attacks in $\mathcal{A}$ if $\forall a \in \mathcal{A}. Var(a) \cap \mathcal{FV}(\Phi) = \varnothing$.*

Proof. Note that the variables used by the active attacker do not occur free in Φ as the intersection is empty (by hypothesis). By Lemma 4.5.1 program P is then robust. □

It is worth noting that we can use this result also for deriving the class of *harmless* active attackers starting from the semantics of the program. Indeed, we can certify that a program is relatively robust wrt. all the active attacks that use low integrity variables which are not free in the formulas representing the private information disclosed before reaching the corresponding hole.

4.6.1 Relative vs Decentralized Robustness

In this section, we show that, in some respects, relative robustness is a more general notion compared to decentralized robustness. The reason is similar to what discussed in Sect. 4.5.4. In a nutshell, we can observe that, once the pair of principals is fixed, the data security levels are also fixed, hence we know which variables are readable and/or modifiable by the attacker q from the point of view of a principal p. We denote by $\mathcal{C}_{p \to q}$ the confidentiality levels and by $\mathcal{I}_{p \leftarrow q}$ the integrity levels for the DLM model. Then, for each variable x, $\mathcal{I}_{p \leftarrow q}(x) = \mathtt{L}$ if p believes that q can modify x, $\mathcal{I}_{p \leftarrow q}(x) = \mathtt{H}$ otherwise. In particular, given a program and a security policy in the DLM fashion, we compute the set of readers and writers for each pair of principals p, q, as done in [74], and then check robustness for each pair using Proposition 4.6.1. This leads to the following generalisation of relative robustness for the DLM model.

Proposition 4.6.2 *Let $P = P_2; [\bullet]; P_1$ be a program (where P_1 is without holes). Let $\Phi = Wlp(P_1, \Phi_0)$. P satisfies decentralized robustness wrt. principals p, q if we have that $\left\{ \; x \;\middle|\; \mathcal{I}_{p \leftarrow q}(x) = \mathtt{L} \; \right\} \cap \mathcal{FV}(\Phi) = \varnothing$.*

Proof. Given a pair of principals (p, q) we compute the set of *readers* and *writers* as for decentralized robustness. Consequently, we have a static labeling of

the program data wrt. confidentiality and integrity. At this point we can apply Lemma 4.5.1 since by assumption no low integrity variable which occurs free in Φ is used by the active attack. Moreover, this holds for all pairs of principals, hence the claim is true. □

This characterization is suitable for client-side languages, for instance Javascript, as it allows to control injection attacks or dynamically loaded third-party code. Suppose we have a web page that accepts advertising adds from different sources with different security requirements and we want to check whether it leaks private information to a malicious attacker. Moreover, suppose that the web page has different trust relations with the domains providing the adds and this is specified in the security policy. Given this information, one can analyze the DOM (Document Object Model) tree and classify each attribute as sensitive and non sensitive with respect to a possible attacker [158]. For example, the *session cookie* may be an attribute to protect wrt. all attackers, while the *history object* may be public to some trusted domains and private to others. At this point we can apply weakest precondition analysis to the web server at the program point where it sends information to any public channel such as the output on web page or the reply information sent as response to a client request. The program holes are the program points where the server receives adds from different clients. Analyzing the formula corresponding to the sensitive information disclosed before embedding the adds, for instance using the *eval()* function in Javascript, we can identify the harmless low integrity variables and certify security modulo (relative to) the programs that manipulate these variables.

Example 4.6.1 *Consider the following Javascript-like code (a modified version of the example in [79]). Lines 3-6 correspond to an add received from a third party to be displayed on the web page. The web site contains a simple function login() which authenticates the users by verifying the username and the password inserted in a form. The function is executed whenever the user clicks on a button (lines 7-16). The function initSettings corresponds to the output channel of the web page and it identifies the server used to authenticate the user, i.e. to send the username and the password.*

```
1.  <script type="javascript">
          // 2: initialization of the output server
2.  initSettings("mysite.com/login.php", 1.0);
          // 3-4-5: definition of the add
3.  <div id="AdNode">
4.    <script src="adserver.com/display.js">
5.  </div>
6.  eval(src)
7.  var login = function() {
8.    var pwd = document.nodes.PasswordTextBox.value;
```

```
9.    var user = document.nodes.UsernameTextBox.value;
11.   var params = "u=" + user + "&p=" + pwd;
          //12: sends the parameters (params) to baseUrl
12.   post(document.settings.baseUrl, params);}
14. </script>
        //15-16:login interface
15. <text id="UsernameTextBox"> <text id="PasswordTextBox">
16. <button id="ButtonLogin" onclick="login()">
```

Now, suppose the ad's code is located at the hole and the public output is the web page together with the result (out : LL*) of post in line 12. The formal parameters of function initSettings (defining the variable baseUrl* : LL*) have low integrity, hence a malicious add could overwrite the parameters and redirect the high confidentiality part of the output of the post (login and password, i.e. user, pwd:* HH*) to the attacker's site. We show how our approach allows to identify these security flaws. First, we compute the weakest precondition of the function login() and obtain the following formula:*

$$
\begin{gathered}
[\bullet] \\
\{baseUrl + user + pwd = a\} \\
var\ pwd = document.nodes.PasswordTextBox.value; \\
var\ user = document.nodes.UsernameTextBox.value; \\
var\ params = "u = " + user + "\&p = " + pwd; \\
\{baseUrl + params = a\} \\
post(document.settings.baseUrl, params) \\
\{out = a\}
\end{gathered}
$$

From the resulting formula we can see that the private information (username and password) is related to the low confidentiality variable baseUrl and therefore the program is not secure. Moreover, the program is not even robust since the variable baseUrl has low integrity and it occurs free before the hole. In particular, a malicious add could send this information to a malicious website and obtain the username and the password. However, the program is robust relative to those fair attacks which are not allowed to manipulate the low integrity variable baseUrl. For decentralized robustness, this corresponds to say that the program is robust wrt. all the pairs (p, q) *such that* p *does not believe that* q *can write the low integrity variable baseUrl.*

4.7 Applications

In this section we present two applications and our approach to capture the security violations. The first example is a secure API function widely used to perform the PIN verification in banks and it is taken from [70]. The attacker is able to modify the low integrity variables and reveal the entire PIN by exploiting an implicit flow in the API. The second example concerns a web application which accepts third

party code and is vulnerable to Cross Site Scripting attacks (XSS) [179]. The attacker tries to steal a session cookie and hijack the user to an evil website. In both examples our analysis is sufficient to spot the security violations.

4.7.1 Secure API Attack

This example concerns the use of secure API to authenticate and authorize a user to access an ATM cash machine. The user inserts the credit card and the PIN code in the machine. The PIN code gets encrypted and travels along the network until it reaches the issuing bank. At this point, a verifying API is executed to validate the trial PIN inserted at the cash machine against the real user PIN. The verifying API, called PIN_V, is the one exploited by the attacker to reveal the real PIN. The real PIN is derived from a PIN derivation key *pdk* and the public data *offset, vdata, dectab*, while the trial PIN comes encrypted with the key k. The two keys, *pdk* and k are pre-loaded in the Hardware Security Modules (HSM) of the bank server and never travel the network. Here is the description of the API, PIN_V.

```
PIN_V(EPB, len, offset, vdata,dectab) {
   x1 := enc_pdk(vdata);
   x2 := left(len, x1);
   x3 := decimalize(dectab, x2);
   x4 := sum_mod10(x3, offset);
   x5 := dec_k(len, EPB);
   if(x4 == x5) then return ("PIN correct");
                else return ("PIN wrong");
}
```

where:

- *len* is the length of real PIN obtained by the encryption of the validation data *vdata* (a kind of user profile) with the PIN derivation key *pdk* (x1), taking the *len* hexadecimal digits (x2), decimalising through *dectab* (x3), and digit-wise summing modulo 10 the offset (x4).
- EPB (Encrypted PIN Block) is the ciphertext containing the trial password encrypted with the key k. The trial PIN is recovered by decrypting EPB with the key k.

The above snippet of code is insecure and there is an attack which discloses the entire PIN just by modifying the low integrity variables *offset* and *dectab* (of type `LL`), and observing the low confidentiality output, namely the I/O behavior of the API method [70].

Example 4.7.1 *Let* $len = 4$, *offset* $= 4732$, $x1 = A47295FDE32A48B1$ *and dectab* $= 9753108642543210$ *which is a substitution function encoding the mapping* $0 \to 9, 1 \to 7, \cdots, F \to 0$. *Moreover, let* $EPB = enc_k(9897)$, *where 9897 is*

the correct PIN. With these parameters PIN_V returns PIN correct.
Indeed, consider $x2 = left(4, A47295FDE32A48B1) = A472$, *and consider* $x3 = decimalize(dectab, A472) = 5165$ *and* $x4 = sum\_mod10(5165, 4732) = 9897$ *which is the same as the trial PIN.*

The attacker first chooses $dectab1 = 97531\underline{1}8642543211$ *where the two 0's have been replaced by 1's. In this way the attacker discovers whether 0 appears in x3. Invoking the API with input dectab1 we obtain the same intermediate and final values, since* $decimalize(dectab1, A472) = decimalize(dectab, A472) = 5165$. *This means that 0 does not appear in x3.*

The attacker proceeds by replacing the 1 in dectab by a 2. Then if $dectab2 = 9753208642543220$, *the attacker obtains that* $decimalize(dectab2, A472) = 5265 \neq decimalize(dectab, A472) = 5165$, *showing the presence of the value in x3. Then,* $x4 = sum\_mod10(5265, 4732) = 9997$ *instead of 9897 returning PIN wrong.*

Now, the attacker knows that the digit 1, occurs in x3 for sure. In order to discover the position and the multiplicity, the attacker varies the offset so that it compensates for the modification of dectab. In particular, the attacker decrements each offset digit by 1 until it finds the digit that forces the API to return PIN correct. For this instance, the possible variations of the offset are: $\underline{3}732, 4\underline{6}32, 47\underline{2}2, 473\underline{1}$ *and the one that succeeds is the offset 4632. Hence, the attacker learns that the second digit of* $x3$ *is 1. Given that the offset is public, he derives the second digit of the user's PIN as* $1 + 7mod10$, *where 7 is the second digit of the initial offset. By iterating this procedure the attacker learns the entire value of PIN.*

We now show how weakest precondition approach captures the security violations in the API. Consider the final formula corresponding to the weakest precondition of the API. Clearly, the program is not secure since the public output (the comparison between the real and the trial password) clearly depends on the high confidentiality variable which contains the real password. Also for robustness, we can note that the sufficient condition is not satisfied since there are low integrity variables, for instance *dectab* and *offset*, which occur free before the hole (supposed to be placed at the input of the API). Indeed, exactly those are the variables used by the attacker for disclosing the PIN.

The security issue can be fixed by using a MAC (Message Authentication Code) security primitive, as proposed in [70]. MACs are used to ensure the integrity of the information received from an untrusted source and they prevent any modification of the data before the API call. Semantically, this means that the variables *dectab* and *offset* can be modified only by the authorized agents. In our approach, the use of a MAC primitive can be modeled by assigning a security level `LH` to the variables *dectab* and *offset*, i.e. by considering them as high integrity. In this way, the problem is now solved since the weakest precondition approach yields a formula where only the high integrity variables occur free. Hence the robustness condition is satisfied.

$$\left\{\begin{array}{c}(sum\_mod10(decimalize(dectab, left(len, enc_p\, dk(vdata))), offset) = dec_k(len, EPB)\\ \wedge a = 1)\vee\\ (sum\_mod10(decimalize(dectab, left(len, enc_p\, dk(vdata))), offset) \neq dec_k(len, EPB)\\ \wedge a = 0)\end{array}\right\}$$

$$x1 := enc_p\, dk(vdata);$$

$$\left\{\begin{array}{c}(sum_m\, od10(decimalize(dectab, left(len, x1)), offset) = dec_k(len, EPB) \wedge a = 1)\vee\\ (sum_m\, od10(decimalize(dectab, left(len, x1)), offset) \neq dec_k(len, EPB) \wedge a = 0)\end{array}\right\}$$

$$x2 := left(len, x1);$$

$$\left\{\begin{array}{c}(sum\_mod10(decimalize(dectab, x2), offset) = dec_k(len, EPB) \wedge a = 1)\vee\\ (sum\_mod10(decimalize(dectab, x2), offset) \neq dec_k(len, EPB) \wedge a = 0)\end{array}\right\}$$

$$x3 := decimalize(dectab, x2);$$

$$\left\{\begin{array}{c}(sum\_mod10(x3, offset) = dec_k(len, EPB) \wedge a = 1)\vee\\ (sum\_mod10(x3, offset) \neq dec_k(len, EPB) \wedge a = 0)\end{array}\right\}$$

$$x4 := sum\_mod10(x3, offset);$$

$$\{(x_4 = dec_k(len, EPB) \wedge a = 1) \vee (x_4 \neq dec_k(len, EPB) \wedge a = 0)\}$$

$$x5 := dec_k(len, EPB);$$

$$\{(x_4 = x_5 \wedge a = 1) \vee (x_4 \neq x_5 \wedge a = 0)\}$$

$$if\ (x_4 == x_5)\ then\ (return\ 1)\ else\ (return\ 0)$$

$$\{l = a\}$$

4.7.2 Cross Site Scripting Attack

Javascript is a very flexible dynamic object-based scripting language running in almost all modern web browsers. The language allows to transfer, parse and run code sent over the network between different web-based applications. While very useful and programmer-friendly, this flexibility comes at a great price since the underlying applications may become vulnerable to code injection attacks. These attacks circumvent the security enforcement mechanism of the browser, namely *the same-origin* policy which prevents a document or script loaded from one origin from getting or setting properties of a document from another origin [158]. Indeed, when the browser receives code from a compromised web page, the code is executed in the context of the website hosting it, therefore, the same-origin policy allows this operation. Afterwards, the malicious code can establish a connection to the attacker's server and transfer sensitive information, a session cookie for instance. The following example shows that language-based security techniques can be used to prevent these kind of attacks.

Suppose a user visits an untrusted web site in order to download a picture and an attacker that has inserted his own malicious Javascript code (Fig. 4.1). This code is executing on the client's browser [179], as described in the following simplified version. The Javascript code snippet in Fig. 4.1 can be used by the attacker to send a user cookie[3] to a web server under the attacker's control.

[3]A cookie is a text string stored by a user's web browser. A cookie consists of one or more

```
/* initialisation of the cookie by the server */
   var cookie = document.cookie;
   var dut;
   if (dut == undefined) {dut = "";}
   while(i<cookie.length) {
       switch(cookie[i]) {
         case 'a': dut += 'a'; break;
         case 'b': dut += 'b'; break;
         ... }
   }
/** dut now contains a copy of cookie;
    when the user clicks on the image, dut is sent
    to the web server under the attacker's control
*/
document.images[0].src = "http://badsite/cookie?" + dut;
```

Figure 4.1: Code creating an XSS vulnerability.

One can easily see that the variable *dut* contains a copy of the user's cookie. This attack circumvents same-origin policy in client browser as it is correctly received after a request to the server where the attacker has previously injected the malicious code. Now we apply our analysis to the above Javascript snippet. Suppose that the variable *cookie* has security type HL and the variable *dut* has security type LL. Also, suppose we rewrite the switch-case statement as a sequence of if-then-else statements and assign to the field *cookie.length* the security type LL.

$$
\begin{array}{c}
[\bullet] \\
\{cookie + dut = res\} \\
while(i < cookie.length)\{ \\
switch(cookie[i])\{ \\
case' a' : dut+ =' a'; break; \\
case' b' : dut+ =' b'; break; \\
...\}\} \\
\{dut = res\}
\end{array}
$$

By observing the final formula we can notice that the confidentiality is violated since there is a (implicit) flow of information from private variable *cookie* towards the public variable *dut*. However, this is the same as the sensitive information disclosed by a passive attacker when the variable *dut* is initialised with the empty

name-value pairs containing bits of information, sent as an HTTP header by a web server to a web browser (client) and then sent back unchanged by the browser each time it accesses that server. It can be used, for example, for authentication.

string. Nevertheless, since *dut* occurs free before the hole, an (active) attacker can exploit *dut* and learn other confidential information from the user. For instance, the attacker may be interested in several properties (attributes) of the *history* object[4] (with security type HL). In this case, an active attack can loop over the elements of the *history* object and use the variable *dut* to reveal all the web pages that the client has had access to. As an example, consider the injection of the code in Fig. 4.2.

```
<script language="JavaScript">
var dut = "";
for (i=0; i<history.length; i++){
    dut = dut + history.previous;
}
</script>
```

Figure 4.2: Malicious code exploiting the XSS vulnerability.

The program violates the robustness condition since the attacker can exploit the low integrity variable *dut*, which occurs free in the formula before the hole, and disclose more confidential information. Moreover the attacker can exploit this vulnerability by inserting the code in Fig. 4.2 right before the malicious code (Fig. 4.1) in the untrusted web page, and learn both the *history* and the *cookie* contents. In general, our approach can be seen as a theoretical model for the existing techniques used to protect the code from XSS attacks [179].

4.8 Related Work

In language-based security, robustness has been addressed by Zdancewic et al. [227, 176]. These papers study a trace-based account of robustness and enforce it using a flow-insensitive type system. The theory is developed on a simple while language, as we do in this paper, moreover, they add to the language a declassification statement which is used to downgrade the security type of the variables at fixed program points (the *where* dimension in [202]). As a result, a program is robust if an active attacker is unable to manipulate the program semantics and force the program to declassify more information than a passive attacker does. The security type system enforces both noninterference and robustness, hence a program is ruled out if neither of the two security properties hold. On the other hand, our semantic approach is different as we model global declassification policies (the *what* dimension in [202]). Moreover, we provide a cleaner characterization of robustness. Namely, the program is robust whenever an active attacker is unable disclose more private information than a passive attacker, although the program under the passive attack does not satisfy

[4]The history object allows to navigate through the history of websites that a browser has visited.

noninterference. Other differences between the two approaches were discussed in Sect. 4.5.4.

The idea of considering the weakest liberal precondition semantics for static certification of program security is borrowed from [162]. The authors define declassified noninterference as a completeness problem in abstract interpretation where the semantic function corresponds to the *Wlp* semantics. However the paper considers only passive attackers. Moreover, the idea of computing *Wlp* wrt. first order formulas is novel.

Decentralized robustness [74] expresses program robustness in the context of the decentralized label model and uses a security type system to enforce it statically. In this paper we showed that decentralized robustness can be modeled using our notion of relative robustness, as discussed in Sect. 4.6.1.

Language-based techniques for security are increasingly being applied to client-side web languages such as Javascript to prevent web attacks [79, 179]. They usually combine static and dynamic analysis to enforce information flow properties such as noninterference. Our idea of putting robustness in the context of Javascript, to the best of our knowledge, is novel and it can be considered as a security model for the language. In particular, the security type `HL` can be assigned the code injected by an attacker, who may know that a certain variable name exists (a variable *password* for instance), but doesn't know its value.

4.9 Conclusions

In this paper, we addressed an important notion in the area of language-based security, namely robustness. Robustness applies to programs that run in environment with untrusted components. This fact is modeled by fixed program points, called holes, where an attacker can insert the untrusted code. Then, the program is robust if an active attacker does not disclose more private information than a passive one. Different active attacks can release different properties of the private data. As the total number of attacks may be infinite, it is impossible to find the most harmful attack for a given program. Therefore, we have provided sufficient conditions that enforce robustness wrt. unfair attacks (using `LL` and `HL` variables). Moreover, we have considered robustness in two different semantic models, the I/O and the trace semantics. We also introduced the notion of relative robustness which is a relaxation of robustness to deal with restricted classes of attacks. Finally, we analysed two case studies: the security API function for PIN verification and the code vulnerable to XSS attacks.

The analysis we performed in this paper results very interesting both from the theoretical and the practical point of view. The semantic condition of robustness addresses the problem of systematic transformations of programs that preserve interesting extensional properties, robustness for instance. The abstract interpretation framework can be used to reason about these security properties. On the other hand, the approach can be a good remedy to the lack of precise static analysis

for more complex applications.

However, this line of work opens up new challenges and much more remains to be done. First, we need to implement the algorithm for static certification of robust programs. That is, given a program we need to effectively (and automatically) compute whether the program is robust. It would also be interesting to characterize the classes of attacks that have the same effect on the disclosure of private information, namely that disclose the same property of private inputs. In this way, we can hope to find a finite number of these attack classes. Second, this work can be generalised to deal with abstract active attackers. Namely, as it happens for abstract noninterference, one can consider attackers modifying properties of low integrity data. Third, we plan to extend our approach to different attacker models such as concurrent attackers or attackers able to erase parts of the program code. Off we go.

Part II

Verification

Chapter 5

ENCoVer: Symbolic Exploration for Information Flow Security

Musard Balliu and Mads Dam and Gurvan Le Guernic

Abstract

We address the problem of program verification for information flow policies by means of symbolic execution and model checking. Noninterference-like security policies are formalized using epistemic logic. We show how the policies can be accurately verified using a combination of concolic testing and SMT solving. As we demonstrate, many scenarios considered tricky in the literature can be solved precisely using the proposed approach. This is confirmed by experiments performed with ENCoVer, a tool based on Java PathFinder and Z3, which we have developed for epistemic noninterference concolic verification.

5.1 Introduction

Information flow security concerns the problem of determining and controlling the nature of information flowing to and from different components of a system. For confidentiality, sensitive information must be prevented from flowing to public destinations, and dually, for integrity, untrusted information must be prevented from affecting, or flowing to, data that needs to be protected. In the possibilistic setting studied here the key property used to model (absence of) information flow is noninterference [122]. Noninterference ensures that the view of an unlicensed observer of the program executions is unaffected by the secret inputs. In a language-based setting, this implies that any two executions having the same public inputs, and possibly different private inputs, produce the same public outputs. Vanilla noninterference turns out to be over-restrictive for many applications, therefore, a controlled

release of private information is usually necessary [197]. This operation is known as *declassification* or downgrading and can be modeled by means of a predicate ϕ over initial private inputs. The idea originates from selective dependency of Cohen [87] and requires that all executions started with initial inputs that satisfy ϕ, should produce the same public observations.

Epistemic logic, the logic of knowledge, provides a clean and intuitive tool for modeling different information flow policies, including noninterference and many variants of declassification, as showed in a number of recent works [38, 128, 28, 45]. The knowledge of an attacker that is in possession of the program text and has partial view of program executions, e.g. by receiving some outputs, can be defined as a partition of the set of secret inputs that determines the observed outputs. This partition corresponds to the properties of secret inputs disclosed by the program. The desired security policy, e.g. some noninterference or declassification property, gives rise to another partition of secret inputs, the property of secret inputs allowed to flow to the observer. Comparing these two partitions determines whether the program meets the security policy. In epistemic logic, the observer's knowledge is expressed in terms of knowledge operator $K\phi$, meaning that the observer knows property ϕ i.e. ϕ is true in all states that are possible given the observer's current state [38, 112]. Intuitively, $K\phi$ holds for all formulas ϕ that induce a partition which is less discriminating (included into) than the one induced by the observed outputs.

Many verification techniques have been proposed for checking information flow properties, including static and dynamic analyses [197]. Security type systems [218, 139] is the dominant technique, but other techniques have been explored as well, including dependency analysis [14], program logics [45], abstract interpretations [119], axiomatic approaches [21], program slicing [220] and so on. Most verification approaches for noninterference-like policies, type systems in particular, enforce noninterference by separating the secret and public computations, and as a consequence any interaction between the secret and public computations, even a benign or corrective one, deems the program as insecure. This increases the number of false positives and limits applicability. Other techniques are based on semantical reasoning and are often computationally expensive or even undecidable. The verification approach proposed in this paper is exclusively tailored to end-to-end verification of noninterference and declassification by means of off-the-shelf epistemic model checkers and SMT solvers. Thereby, the approach is both sound and complete with respect to verification in the underlying (bounded) program model. Other works on model checking-based verification of security properties are considered in a later section [71, 19, 128, 215].

In this paper, concolic testing, a mix of concrete and symbolic execution, is used to extract a bounded model of program runtime behavior [121, 146, 205, 185]. This model is subsequently verified against the target security properties, expressed in epistemic logic, by means of an epistemic model checker. Due to the size of the input data domain epistemic model checking can, however, be extremely inefficient or even infeasible. To address this, an alternative approach is proposed whereby the model

checking problem is transformed to a first order logic formula. Due to the shape of epistemic formulas for noninterference and declassification, the transformation produces a formula which only contains existential quantifiers, thereby an SMT solver can be used to perform the checking efficiently.

We have implemented the verification approach described above in a tool prototype, ENCoVer. The prototype is an extension of Java PathFinder, a software model checker developed at NASA [184]. ENCoVer takes as input a program written in Java and a security policy and generates a *symbolic output tree*, which encodes conditions on program inputs that produce output observations. The symbolic output tree is used in two ways. First, it is combined with the security policy to generate an SMT formula which is subsequently verified with Z3, a state-of-art SMT solver [98] and, secondly, as an alternative, it is used to generate an input file for the epistemic model checker MCMAS [157]. The performance of ENCoVer is evaluated on a main case study involving multiple parties accessing a joint store of tax records, as well as on several smaller, but delicate, examples.
In summary, the main contributions of the paper are

- A framework for concolic verification of information flow properties based on epistemic logic
- A symbolic model checking algorithm for noninterference-like policies
- Formal correctness proofs of the model transformations involved
- A tool prototype, ENCoVer, implementing the verification techniques
- Evaluation of the ENCoVer tool on a non-trivial case study

The paper starts by presenting the background context (Sect. 5.2) — including the computational model, the epistemic logic and the security properties of interest — needed to expose the concolic testing based algorithm used to extract the program model which is presented with the associated proofs in Sect. 5.3. Information flow related epistemic formulas can be verified on this model using either an epistemic model checker (Sect. 5.4.1) or an SMT solver (Sect. 5.4.2). This approach has been implemented in a prototype, ENCoVer, and applied to a case study (Sect. 5.5) whose evaluation results are presented in Sect. 5.6. Related work is addressed before concluding in Sect. 5.8.

5.2 Preliminaries

In this section we introduce the computational model based on labelled state transition systems, and an epistemic logic which is used to specify security properties over the computational model. A more detailed discussion of the information flow properties that can be characterized by this logic can be found in [38].

5.2.1 Computational Model

A *labelled transition system* $STS = (\mathcal{S}, Act, \mathcal{T}, \mathcal{S}_0)$ consists of a set of states $\sigma \in \mathcal{S}$, resp. actions $\alpha \in Act$, a labelled transition relation $\mathcal{T} \subseteq \mathcal{S} \times Act \times \mathcal{S}$, and a set of initial states $\mathcal{S}_0 \subseteq \mathcal{S}$. The set of actions contains a neutral element ϵ representing inaction. Other elements of Act are assumed to be observable, and represent interactions with the environment, for instance as inputs or outputs. The transition relation $\sigma \xrightarrow{\alpha} \sigma'$ states that by taking one execution step in state $\sigma \in \mathcal{S}$ the execution generates the action $\alpha \in Act$ and the new state is $\sigma' \in \mathcal{S}$. We write $\sigma \rightarrow \sigma'$ for $\sigma \xrightarrow{\epsilon} \sigma'$. An *execution* is a finite sequence of execution states

$$\pi = \sigma_0 \xrightarrow{\alpha_0} \sigma_1 \xrightarrow{\alpha_1} \cdots \xrightarrow{\alpha_{n-1}} \sigma_n \tag{5.1}$$

where $\sigma_0 \in \mathcal{S}_0$ and $\sigma_i \xrightarrow{\alpha_i} \sigma_{i+1} \in \mathcal{T}$ for all $0 \leq i < n$. The length, $len(\pi)$, of π is n. An *execution point* is a pair (π, i) where $0 \leq i \leq len(\pi)$. The i'th execution state is $\sigma(\pi, i) = \sigma_i$. We write $trunc(\pi, i)$ for the prefix of π up to and including σ_i.

The observable part of the system is modeled by a function *trace* mapping executions to sequences of observations.

Definition 5.2.1 (Trace) *A* trace τ *is sequence of observable actions. For* π *as in (5.1), the trace of* π *up to point* $i : 0 \leq i \leq n$ *is the sequence* $trace(\pi, i)$ *of actions* α_j *where* $0 \leq j < i$ *and* $\alpha_j \neq \epsilon$.

We write $trace(\pi)$ for $trace(\pi, len(\pi))$.

In a more general setting, $trace(\pi, i)$ can span from the truncation function $trunc(\pi, i)$ for the strongest observer able to see all the internal computation, to the function returning the last action generated for a weak memoryless observer. In the remainder of this paper, we use the function *trace* given in Def. 5.2.1. This definition corresponds to the perfect recall observer, i.e. only able to observe actions and having full memory of past observations.

Finally, a model $\mathcal{M}_{STS}$ (or simply $\mathcal{M}$) is a set of executions induced by a state transition system STS. Normally we take as a model the set of all executions originating from some set of initial states $\mathcal{S}_0$.

5.2.2 Interpreted Systems

The computational model can be associated with an *interpreted system* [112]. In our two agent case, an interpreted system consists of an environment agent E and an agent under observation A, which interact over the course of a computation. Each agent i can be in local state L_i and perform action ACT_i. A protocol $P_i \subseteq L_i \times ACT_i$ selects actions depending on the current local state and an evolution function $t_i \subseteq L_i \times ACT \times L_i$ describes how agent i moves to a new state depending on a joint action $ACT = \times_i ACT_i$ performed by system agents. The product of evolution and protocol functions determine how the system changes its global state. In particular, a *global state* is the product of agent's local states, $g = (L_E, L_A)$.

Agent A has a *local state* $L_A = trace(\pi, i)$ that records the sequence of actions that have occurred when the environment E was in state $L_E = trunc(\pi, i)$. A global state $g = (L_A, L_E)$ describes the system at a given point in time. In our case, as we will see in Sect. 5.4, agent A performs no actions, while agent E emits observable actions. An execution π induces a sequence of global states, called *runs* r, such that for all execution points π, i, $r(\pi, i) = (trace(\pi, i), trunc(\pi, i))$. The initial state set I_0 is a subset of global states G, where $g_0 \in I_0$ and $g_0 = (\epsilon, trunc(\pi, 0))$ for some $\pi \in \mathcal{M}$. Finally an *evaluation* function $V : G \to \wp(AP)$ defines, for every global state $g \in G$, the subset of atomic propositions $V(g) \in \wp(AP)$ holding in g.

Definition 5.2.2 (Interpreted System) *An interpreted system $\mathcal{I}$ over two agents $Ag = \{E, A\}$, a set of atomic propositions AP and a non empty initial state I_0 is a tuple*

$$\mathcal{I} = \langle \{L_i\}_{i \in Ag}, \{ACT_i\}_{i \in Ag}, \{P_i\}_{i \in Ag}, \{t_i\}_{i \in Ag}, I_0, V \rangle$$

To define knowledge, we associate an interpreted system $\mathcal{I}$ with a *Kripke structure* $\mathcal{M}_\mathcal{I} = (G, V, K_A)$ where G and V are defined as before and K_A is a binary relation over G. In particular, K_A defines the *indistinguishability* relation for agent A, which is an equivalence relation among global states from the point of view A. Two global states $g_1, g_2 \in K_A$ are indistinguishable iff they define the same trace τ. Next we introduce a logic where a formula ϕ is known to agent A at global state g if that ϕ is true for all global states in the K_A relation with g.

5.2.3 Epistemic Propositional Logic

We now present a very simple logic that will be used to reason about properties in the model described previously. Let *Val* be a domain of values c, *Ide* a finite set of (program) identifiers x, and u, v range over first order variables. Arithmetic and boolean expressions use values, identifiers and variables along with some set of arithmetic and boolean operators, left unspecified for now. The language $\mathcal{L}_{KU}$ of epistemic first-order formulas ϕ, ψ is:

$$\phi, \psi ::= b \mid \forall u.\phi \mid \phi \to \psi \mid \neg\phi \mid K\phi$$

The logic contains primitive predicates b over identifiers x and first order variables u. Program identifiers are interpreted in the initial state and first order variables are rigid i.e. independent of the state. The formula $\forall u.\phi$ universally quantifies over rigid variables. The operator K is the epistemic knowledge operator. A formula $K\phi$ holds in an execution point iff ϕ holds in any execution point epistemically equivalent to the current one, i.e. ϕ is true in all execution points having the same trace as current execution point. Various connectives are definable in $\mathcal{L}_{KU}$ including the epistemic possibility operator $L\phi = \neg(K(\neg\phi))$ meaning that ϕ holds in at least one epistemically equivalent execution point.
The semantics is given in terms of satisfaction relation $\mathcal{M}, \pi, i \models \phi$ at execution points (π, i) in $\mathcal{M}$. If the model $\mathcal{M}$ is clear from the context we write $\pi, i \models \phi$ for

$\mathcal{M}, \pi, i \models \phi$. An execution π satisfies a formula ϕ, $\pi \models \phi$, if for all $0 \leq i \leq \mathit{len}(\pi)$, $\pi, i \models \phi$. A model $\mathcal{M}$ satisfies formula ϕ, $\mathcal{M} \models \phi$, iff for all $\pi \in \mathcal{M}$, $\pi \models \phi$. In the remainder of this paper we take as model the set of executions generated by some program P as detailed in Sect. 5.3. A state is a finite map $\sigma : x \mapsto c$, and $\sigma(e)$ denotes the value of formula or expression e in state σ. The observable actions are output values belonging to $\mathit{Act} = \{\texttt{out}(c) \mid c \in \mathit{Val}\}$. Below we report a few cases of satisfaction relation. Other cases work as expected [38].

- $\pi, i \models b$ iff $\sigma(\pi, 0)(b)$
- $\pi, i \models \forall u.\phi$ iff for all $c \in \mathit{Val}$ $\pi, i \models \phi[u \mapsto c]$
- $\pi, i \models K\phi$ iff for all execution points π', i' such that $\mathit{trace}(\pi, i) = \mathit{trace}(\pi', i')$, $\pi', i' \models \phi$
- $\pi, i \models L\phi$ iff there exists an execution point π', i' such that $\mathit{trace}(\pi, i) = \mathit{trace}(\pi', i')$ and $\pi', i' \models \phi$

It is worth noting that the satisfaction relation over primitive predicates only considers the initial value of identifiers. The reason is that we are interested in verifying properties that depend only on the initial assignment to program identifiers.

Example 5.2.1 *Let $\mathcal{M}$ be the model of program P with input identifier h. The initial value of h should remain secret to the observer who knows the program text and can see the program outputs. Let $b(h)$ be a primitive predicate over identifier h.*

1. *$\mathcal{M} \models \neg K(b(h))$: Model $\mathcal{M}$ satisfies the formula iff for all execution points (π, i), the observer can not tell whether $b(h)$ holds. Namely, for all points that are epistemically possible, there exists at least one, say π', i', such that $\mathit{trace}(\pi, i) = \mathit{trace}(\pi', i')$ and $\pi', i' \not\models b(h)$. Hence the system keeps property $b(h)$ secret, which is known as* opacity *[66].*
2. *$\mathcal{M} \models L(b(h)) \wedge L(\neg b(h))$: Model $\mathcal{M}$ satisfies the formula iff for all execution points (π, i), both $b(h)$ and its negation are possible i.e. there exist π', i', π'', i'' where $\mathit{trace}(\pi, i) = \mathit{trace}(\pi', i') = \mathit{trace}(\pi'', i'')$ and $\pi', i' \models b(h)$ and $\pi'', i'' \models \neg b(h)$. Hence the observer is unable to deduce any information about the property (or its negation) by looking at the sequences of outputs. This security property is known as* secrecy *[128].*

5.2.4 Noninterference and Declassification

The absence of illegal information flows in a system is usually expressed as a noninterference security condition [122]. In a possibilistic setting with a two-level security lattice only, noninterference requires that *high/secret* input values do not influence *low/public* output values. In this paper high inputs correspond to the initial values of secret identifiers and low outputs correspond to the traces defined in Section

5.2.1. We write $\sigma_1 \approx_{\vec{x}} \sigma_2$ if two states σ_1 and σ_2 are equivalent with regard to a set of identifiers $\vec{x}$, i.e. $\forall x \in \vec{x}.\, \sigma_1(x) = \sigma_2(x)$. Consider now a set of low identifiers $\vec{l}$, whose initial value is known a priori, and a set of high identifiers $\vec{h}$. A program P satisfies *noninterference* (NI) iff for any two executions starting with equal initial values for $\vec{l}$ the following condition holds.

$$\forall \pi_1, \pi_2 \in \mathcal{M}_P.\, \sigma(\pi_1, 0) \approx_{\vec{l}} \sigma(\pi_2, 0) \Rightarrow \mathit{trace}(\pi_1) = \mathit{trace}(\pi_2)$$

NI can be characterized using the epistemic logic $\mathcal{L}_{KU}$. A program P satisfies *absence of knowledge (AK)* if its associated model M_P satisfies the following formula.

$$\mathcal{M}_P \models\ \forall \vec{v}.\, \vec{l} = \vec{v} \rightarrow \forall \vec{u}.\, L(\vec{l} = \vec{v}\ \wedge\ \vec{h} = \vec{u})$$

That is, any initial high input must be possible among the executions having the same trace and the same initial low inputs.

Noninterference turns out to be an over-restrictive policy for many applications. A controlled release of secret information is necessary in many real software applications. This feature is known as *declassification* or downgrading and remains a challenge in information flow security [197]. One way of modeling declassification is by means of a predicate ϕ, over initial values, which expresses the property to declassify. Then the security condition states that all secret inputs having the same property ϕ should not be distinguished by the external observer. Let $\sigma_1 \approx_\phi \sigma_2$ denote equivalent states according to the declassification policy ϕ i.e. $\sigma_1(\phi) = \sigma_2(\phi)$. A program P satisfies *noninterference modulo declassification (NID)* ϕ if:

$$\forall \pi_1, \pi_2 \in \mathcal{M}_P.\, (\sigma(\pi_1, 0) \approx_{\vec{l}} \sigma(\pi_2, 0) \wedge \sigma(\pi_1, 0) \approx_\phi \sigma(\pi_2, 0)) \Rightarrow \mathit{trace}(\pi_1) = \mathit{trace}(\pi_2)$$

The definition of NID specifies that any initial state having the same low input values and agreeing on ϕ should produce the same output trace. Let ϕ be the declassification policy. A program P satisfies *absence of knowledge modulo declassification (AKD)* ϕ if:

$$\mathcal{M}_P \models \forall \vec{v}_1, \vec{u}_1.(\vec{l} = \vec{v}_1 \wedge \vec{h} = \vec{u}_1) \rightarrow \forall \vec{u}_2.(\phi(\vec{v}_1, \vec{u}_1) \leftrightarrow \phi(\vec{v}_1, \vec{u}_2)) \rightarrow L(\vec{l} = \vec{v}_1\ \wedge\ \vec{h} = \vec{u}_2)$$

The semantical definition *NID* is proved to be equivalent to its epistemic characterization *AKD* in [38]. The following example will walk us through presenting the verification approach in the subsequent sections of the paper.

Example 5.2.2 *Consider the program P with high identifier secret ranging over non-negative integers up to a fixed constant max.*

$$P ::= \left[\begin{array}{l} i := 0; \\ \texttt{if}\ (\mathit{secret} < 0)\ \texttt{then}\ \mathit{secret} = 0; \\ \texttt{if}(\mathit{secret} > \mathit{max})\ \texttt{then}\ \mathit{secret} = \mathit{max}; \\ \texttt{while}\ (i < \mathit{secret})\ \texttt{do out}(i++); \\ \texttt{while}\ (\mathit{secret} < \mathit{max})\ \texttt{do out}(\mathit{secret}++); \end{array}\right.$$

Clearly P is noninterfering since it outputs (statement `out`*) the same sequence of numbers for any choice of secret, yet the example is tricky to verify for most approaches in the literature, and it illustrates well the complications regarding mixed data and control flow our approach needs to handle. Too see that P is noninterfering, consider the model $\mathcal{M}$ of P and the corresponding AK formula $\phi = \forall u.L(secret = u)$. We show that $\mathcal{M} \models \phi$. Let $\pi \in \mathcal{M}$ be an execution originating from state $\sigma(\pi, 0) = (max_0, i_0, secret_0)$ and $0 \leq j \leq len(\pi)$. For all values c such that $secret = c$, there exist π', j' originating from state $\sigma(\pi', 0) = (max_0, i_0, c)$ such that $trace(\pi, j) = trace(\pi', j')$. In fact, all executions output the sequence of non-negative integers up to max_0.*

5.3 Program Analysis by Concolic Testing

In this section we present the formal underpinnings of the approach we use for extracting the program model and checking formulas in $\mathcal{L}_{KU}$. The main idea is to start from the flow graph of the source program, extract, by means of concrete and symbolic execution (concolic testing), an abstract model, and then use an epistemic model checker or an SMT solver to verify formulas over this model.

We impose some constraints to make the construction tractable. First we assume that all inputs from the external environment are read at the start of program execution. This restriction rules out reactive programs that receive external inputs during execution. However, provided the original program can be transformed, one can anticipate reading inputs in the beginning of execution in many cases. Secondly, we assume a bounded model of runtime behavior, hence programs always terminate, loops can be unfolded, method calls or exception handlers can be inlined in the main method body and so on. This allows to present source programs in the form of execution trees defined as follows.

Definition 5.3.1 (Basic Block, BB) *A* basic block *is a portion of sequential code (without jumps) of the following type:*

- Simple Basic Block (SBB)*: A sequence of assignments $b_1; b_2 \cdots b_n$*
- Output Basic Block (OBB)*: A single output expression* `out`*(exp), for some expression exp*

Definition 5.3.2 (Execution Tree, ET) *An* execution tree *is a directed labelled tree $T = (B, E, C, L, \text{Start})$ such that*

- *B is a set of nodes n labelled by basic blocks $B(n)$*
- *$E \subseteq B \times B$ is a set of control flow edges*
- *C is a set of branch conditions, boolean expressions over program identifiers*
- *$L : E \mapsto C$ is a mapping from edges to branch conditions*

- Start $\in B$ *is the root node*

For convenience we extend T with a special node *End*, in order to make terminal states explicit in the construction. To this end we require that $\bigvee\{L(n, n') \mid n' \in B\}$ is a tautology for each node $n \in B - \{End\}$, something which is easily achieved. This allows attention to be restricted to executions that start at the *Start* node, follows the ET control structure in the obvious way, and end at the *End* node. For deterministic programs each initial state σ_0 determines a unique such execution π with $\sigma(\pi, 0) = \sigma_0$. In general a fixed initial state can determine a set of executions due to different thread schedulers as well as possible internal nondeterminism.

Definition 5.3.3 (ET path) *Given an execution tree T, a path Π is a sequence of consecutive basic blocks from the node* Start *to the node* End*, connected by labelled edges in E. The set $Paths(T)$ is the set of all paths in T. The length, $len(\Pi)$, is the number of basic blocks in Π.*

Definition 5.3.4 (ET model) *A model of an ET T is the set of all executions of T beginning in initial state σ_0 and following a path $\Pi \in Paths(T)$.*

Example 5.3.1 *The execution tree corresponding to the program in Example 5.2.2 is shown in Fig. 5.1. Here, for compactness, we depict the ET as a graph, the tree representation is easily derived by unfolding the loops.*

Execution trees are analyzed using concolic testing to produce an abstract version called a *symbolic output tree*. Concolic testing is a software verification technique that combines executions on concrete and symbolic values [121, 205, 185]. A concrete execution is a normal run of the program from an initial input state. In symbolic execution unknown input is represented as symbolic values and the output is computed as a function of these values [146]. Consequently, the program state is also symbolic and it includes expressions over symbolic values of program identifiers.

States in the symbolic output trees are associated with a *path condition* which represents a boolean predicate on initial inputs and defines the constraints these inputs must satisfy so that a concrete execution follows that path. Symbolic execution can be viewed as a predicate transformer semantics that represents programs as relations between logical formulas and it is tightly related to strongest postcondition computations [223].

A concolic testing algorithm does the following in a loop until all ET paths are explored: it starts with concrete and symbolic values for input variables and executes the program concolically by collecting at each step path conditions. These conditions are later used to generate, by means of a constraint solver, a new input that explores a different path. When an output statement is reached, the corresponding output expression is also evaluated in the symbolic state. The symbolic output tree represents conditions on initial inputs that direct the program to an output statement. This is done by saving the path conditions and the output expressions for all reachable basic blocks.

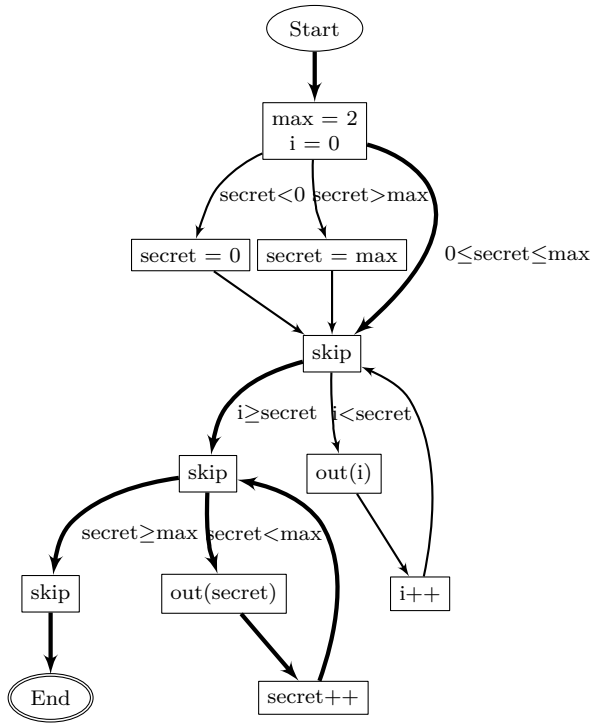


Figure 5.1: Execution Tree (represented as a graph due to lack of space)

Definition 5.3.5 (SOT) *A Symbolic Output Tree is an ET which only contains output basic blocks.*

The following algorithm describes how the symbolic execution part of the analysis extracts the SOT from the ET. The concrete executions are not reported in the algorithm as they do not directly participate in the construction of the SOT. Algorithm 1 uses the procedure DFSVisit to visit the ET and build the SOT on the fly. The input is an initial ET T and the output is the corresponding SOT S. The algorithm creates an SOT S containing a *Start* and an *End* node (line 1) and then calls the procedure DFSVisit with input parameters the initial nodes of T and S, the symbolic state *Sym* generated by function *InitSym* (a map from input identifiers in T to symbolic values), and the path condition Pc (initially set to *true*), respectively (line 2). Moreover *CurrT.Children* are the immediate successors of node *CurrT*, SAT(Pc) checks whether formula Pc is satisfiable, Eval(EF, *Sym*) evaluates an expression or a formula EF in the symbolic state *Sym*, Add(A, a) adds a node a to a set A, and finally SP(*B.Stat*, *Sym*) computes the strongest postcondition for the sequence of statements in *B.Stat* and *Sym*.

The algorithm visits all basic blocks in the tree. If the basic block is a simple

Algorithm 1 ET to SOT

INPUT: ET T
OUTPUT: SOT S
1. S := new SOT()
2. **Call** DFSVisit($T.Start$, $S.Start$, $InitSym$, $true$)

DFSVisit(ET node $CurrT$, SOT node $CurrS$,
Symbolic state Sym, Path condition Pc)
1. **For** B in $CurrT.Children$
2. Pc := Eval(L($CurrT$, B), Sym) $\wedge$ Pc
3. **If** SAT(Pc)
4. **If** B is OBB
5. $SotN$:= new OBB(Eval($B.Out$, Sym))
6. Add($CurrS.Children$, $SotN$)
7. L($CurrS$, $SotN$) := Pc
8. $CurrS$:= $SotN$
9. **Else If** B is SBB
10. Sym := SP($B.Stat$, Sym)
11. **Else**
12. Add($CurrS.Children$, $S.End$)
13. DFSVisit(B, $CurrS$, Sym, Pc)

basic block, the algorithm updates the symbolic state by computing the *strongest postconditions* (line 10). If the basic block is an output basic block, it evaluates the output expression in the current symbolic state and saves the result in a new SOT node (line 5), connects the nodes with an edge labelled by current Pc and updates the current node (line 6-8). Otherwise, an *End* node has been reached, hence, the current node is connected (line 12). An SMT solver is used to determine whether the conjunction of the path condition with the edge condition evaluated in the symbolic state is satisfiable (line 2-3). If this is the case, then there exist inputs that can explore that path, thus the algorithm continues with the analysis of the basic block (line 4-13). Otherwise, if the formula is unsatisfiable, the path will never be taken, so the algorithm backtracks and explores another edge condition (line 1). The analysis continues until all reachable basic blocks have been explored and the corresponding symbolic output tree has been constructed. The symbolic states are saved at each step of the analysis, hence it is possible to restore the right one during the backtracking phase of the algorithm.

Example 5.3.2 *Figure 5.2 shows the symbolic output tree generated by Alg. 1 on execution tree in Fig. 5.1. Let* $Sym = [secret \mapsto \alpha]$ *and* $Pc := true$ *be the initial symbolic state and path condition, respectively. Suppose Alg. 1 chooses to analyse*

first the path depicted in bold arrows in Fig. 5.1. The first SBB is reached and the local variables max and i are added to $Sym^1 = [secret \mapsto \alpha, i \mapsto 0, max \mapsto 2]$*, while Pc remains unchanged as the edge condition, i.e. true, evaluated in* Sym^1 *is the same. The next two basic blocks only update the path condition to* $Pc^1 := (0 \leq \alpha \leq max \wedge i \geq \alpha)$ *since* skip *has no effect on the symbolic state. Afterwards the path condition becomes* $Pc^2 := (0 \leq \alpha \leq max \wedge i \geq \alpha \wedge Eval((secret < max), Sym^1))$ *which evaluates to* $(\alpha = 0)$*. The corresponding OBB statement,* $out(secret)$*, is then evaluated in* Sym^1 *and a new OBB is added to SOT with output expression* $Eval(secret, Sym^1) = \alpha$*. The next SBB produces* $Sym^2 := Sym^1[secret \mapsto \alpha + 1]$*, as* $SP(secret++, Sym^1) = Sym^1[secret \mapsto Sym^1(secret) + 1]$*. The path condition remains unchanged as the edge condition was the constant true. The DFS analysis enters the loop one more iteration, creates the OBB node with* $Eval(secret, Sym^2) = \alpha + 1$ *and yields* $Sym^3 := Sym^2[secret \mapsto \alpha + 2]$ *and* $Pc^2 := (\alpha = 0)$*. At this point the condition* $(\alpha = 0 \wedge \alpha + 2 \geq max)$ *becomes true and the algorithm starts the backtracking phase. The bold path in Fig. 5.2 corresponds the path created by the DFS analysis explained here.*

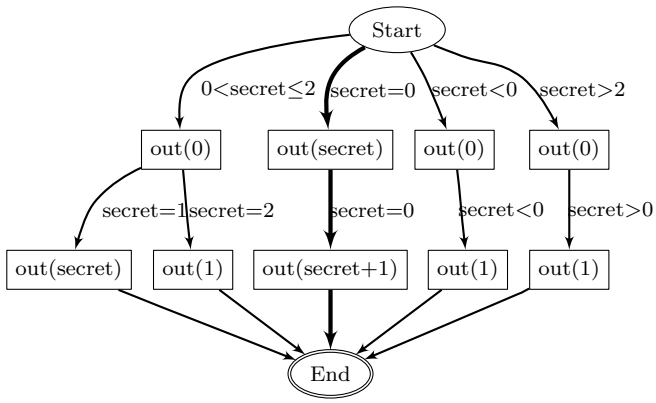


Figure 5.2: Symbolic Output Tree

5.3.1 Formal Correctness

We now move to proving correctness of the approach and showing that the abstraction generated by the SOT is complete with respect to the formulas in $\mathcal{L}_{KU}$. As we show in Lemma 5.3.1 this boils down to proving the equivalence between *pre-traces* generated by the ET and *executions* generated by the SOT.

Definition 5.3.6 (ET execution) *Let* C *be a boolean expression over identifiers and* T *an ET. Then* $Exec(C, T)$ *is the set of all executions* π *in* T *where* $\sigma(\pi, 0) \models C$*. We abbreviate* $Exec(true, T)$ *as* $Exec(T)$*.*

Definition 5.3.7 (ET pre-trace) *Let π be an execution in an ET T where $\pi = \sigma_0 \xrightarrow{\alpha_0} \sigma_1 \xrightarrow{\alpha_1} \sigma_2 \xrightarrow{\alpha_2} \cdots \xrightarrow{\alpha_n} \sigma_n$. Then a* pre-trace *is the execution*

$$ptrace(\pi) = \sigma_0 \xrightarrow{\alpha_{i_0}} \sigma_0 \xrightarrow{\alpha_{i_1}} \sigma_0 \xrightarrow{\alpha_{i_2}} \cdots \xrightarrow{\alpha_{i_k}} \sigma_0$$

where $\alpha_{i_j} \neq \epsilon$ and $trace(\pi) = trace(ptrace(\pi))$. Moreover, $ptrace(C, T)$ is the set of pre-traces of $Exec(C, T)$. Similarly $ptrace(T)$ is the set of pre-traces of $Exec(T)$.

A trace consists of the sequence of outputs in the pre-trace and many pre-traces can correspond to the same trace. A pre-trace can be viewed as an execution, hence satisfiability and validity of a formula over $ptrace(E)$ is defined as for the executions. Since the formulas in logic $\mathcal{L}_{KU}$ concern initial input values only, one can prove the following lemma.

Lemma 5.3.1 *Let π be an execution in a model $\mathcal{M}$ and $ptrace(\pi)$ the pre-trace in the corresponding pre-trace model $ptrace(\mathcal{M})$. Then, for all formula ϕ in $\mathcal{L}_{KU}$*

$$\mathcal{M}, \pi \models \phi \Leftrightarrow ptrace(\mathcal{M}), ptrace(\pi) \models \phi$$

Proof. Induction on structure of formula ϕ. Suppose $\phi = K\phi'$: We get $\pi \models \phi$ iff for all π' such that $trace(\pi) = trace(\pi')$, $\pi' \models \phi'$. But, by induction hypothesis, we know $ptrace(\pi') \models \phi'$, hence we're done. Suppose $\phi = b$: Then $\pi \models b$ iff $\sigma(\pi, 0) \models b$. But also $ptrace(\pi) \models b$ iff $\sigma(\pi, 0) \models b$. Other cases are equally trivial and the other direction holds as the logic is closed under negation. □

An SOT is an ET, therefore the executions are defined in the same manner. One can easily show that all executions generated by SOT are pre-traces. The next step is to prove that an ET and the corresponding SOT define the same set of pre-traces. Then, one can prove properties expressed in $\mathcal{L}_{KU}$ in the SOT model, which by Lemma 5.3.1 will hold in the original ET model.

Lemma 5.3.2 *Let σ_0 be a concrete program state and Sym a symbolic state. Then, for all SBBs B^* there exist σ, σ' and Sym' such that*

$$Eval(Sym, \sigma_0) = \sigma \wedge (B^*, \sigma) \rightarrow \sigma' \wedge \\ SP(B^*, Sym) = Sym' \Rightarrow Eval(Sym', \sigma_0) = \sigma'$$

Lemma 5.3.3 *Let C, Pc be two boolean expressions on program identifiers, σ_0, σ two concrete states and Sym a symbolic state. Then,*

$$Eval(Sym, \sigma_0) = \sigma \ \wedge \ \sigma_0 \models Pc \ \wedge \ \sigma \models C \\ \Rightarrow \sigma_0 \models Pc \wedge Eval(C, Sym)$$

Lemma 5.3.4 *Let π be an ET execution and B^* the SBB between states σ_i and σ_j as in the execution.*

$$\pi = \sigma_0 \xrightarrow{\alpha_0} \cdots \sigma_i \xrightarrow{\epsilon} \cdots \xrightarrow{\epsilon} \sigma_j \cdots \xrightarrow{\alpha_{n-1}} \sigma_n$$

Then $ptrace(\pi) = ptrace(\pi')$ where $\pi' = \sigma_0 \xrightarrow{\alpha_0} \cdots \sigma_i \xrightarrow{\epsilon} \sigma_j \cdots \xrightarrow{\alpha_{n-1}} \sigma_n$.

Lemmas 5.3.2 and 5.3.3 state that the path condition and the symbolic state computed by Alg. 1 represent the set of initial states that lead to the program point they are associated with. If σ is a state obtained by evaluating a symbolic state Sym in a state σ_0 that satisfies the path condition Pc, then there exists a concrete program execution starting from σ_0 and reaching state σ. On the other hand Lemma 5.3.4 shows that the program instructions in an SBB can be considered as executed atomically since they will produce the same pre-trace anyway.

Theorem 5.3.1 (ET-SOT pre-trace equivalence) *Let T be an ET and S the corresponding SOT generated by Alg. 1. Then,*

$$ptrace(T) = Exec(S)$$

Proof. [Proof Sketch] We prove inclusion in both directions using previous Lemmas.
($\Rightarrow$) We show that $ptrace(T) \subseteq Exec(S)$ by induction on the length i of an ET execution using Algorithm 1. This can be reduced to induction on length i' of executions π' derived from $\pi \in Exec(T)$ as in Lemma 5.3.4. Intuitively executions π' have the same length as the path in the ET which they correspond to. Let the resulting model be $Exec(T')$ and $Cl(Exec(T'))$ its prefix closure. Then we show that for all $\pi' \in Cl(Exec(T'))$, there exists an (prefix) execution $\pi^* \in Cl(Exec(S))$ and $ptrace(\pi') = \pi^*$. This is done by proving that there exist nodes N_T in the ET, N_S in SOT, Sym and Pc such that (a) π' is an execution from $Start$ to N_T (b) π^* is an execution from $Start$ to N_S (c) Algorithm 1 calls DFSVisit(N_T, N_S, Sym, Pc) and (d) $ptrace(\pi) = \pi^*$ and $\sigma(\pi, len(\pi)) = Eval(Sym, \sigma(\pi, 0))$ and $\sigma(\pi, 0)(Pc)$.
Base case: ($i = 0$) Let $\pi' \in Cl(Exec(T))$ and $len(\pi') = 0$, then $\pi' = \sigma_0$ by definition. Algorithm 1 starts with a symbolic state ($InitSym$ in line 2) when it first creates the SOT node. Hence, any $\pi^* \in Cl(Exec(S))$ with $\sigma(\pi^*, 0) = \sigma_0$ will do. Moreover, DFSVisit($Start$, $Start$, $InitSym$, $true$) is initially called with $N_T = Start$, $N_S = Start$ and $\pi = \pi^* = \sigma_0$ is such an execution. In particular, $ptrace(\pi) = \pi^* = \sigma_0$, $\sigma(\pi, len(\pi)) = Eval(Sym, \sigma(\pi, 0)) = \sigma_0$ and $\sigma(\pi, 0)(Pc) = \sigma(\pi, 0)(true)$ which trivially holds.
Induction: We prove that for all $\pi \in Cl(Exec(T))$ with $len(\pi) = k$, there exists $\pi^* \in Cl(Exec(S))$ and all conditions (a-d) hold. By induction hypothesis, conditions (a-d) hold for the prefix execution of length $k - 1$ of π, say $\pi' \in Cl(Exec(T))$. Let π'^* be the corresponding SOT execution and Algorithm 1 has called DFSVisit(N'_T, N'_S, Sym', Pc'). Then, $ptrace(\pi') = \pi'^*$, $\sigma(\pi', len(\pi')) = Eval(Sym', \sigma(\pi', 0))$ and $\sigma(\pi', 0)(Pc')$ holds. Let now C be the boolean expression associated with the edge from N'_T to N_T and $\sigma(\pi', 0)(C)$, otherwise we are done. There are two possible cases. First suppose N_T is an OBB (with $\texttt{out}(e)$) that outputs $v = \sigma(\pi, len(\pi))(e)$. Since an output action is performed, both execution state and symbolic state remain unchanged, hence $\sigma(\pi, len(\pi)) = Eval(Sym, \sigma(\pi, 0))$ and $Pc = Pc' \wedge Eval(C, Sym')$. Then, by Lemma 5.3.3 also $\sigma(\pi, 0)(Pc)$ holds. The output value is v since $Eval(Eval(e, Sym), \sigma(\pi, 0)) = Eval(e, \sigma(\pi, len(\pi))) = v$.

Otherwise, N_T is an SBB. By applying Lemma 5.3.2 and 5.3.3, similarly it can be shown that the path condition and the symbolic state are computed correctly.
($\Leftarrow$) We prove that $ptrace(T) \supseteq Exec(S)$ if for all executions $\pi^* \in Exec(S)$ there exists $\pi \in Exec(T)$ and $\pi^* = ptrace(\pi)$. The induction hypothesis works as previously. The only difference is that a single transition in SOT can correspond to an arbitrary but finite number of SBBs followed by one OBB in the ET. In that case the claim is proved by applying Lemma 5.3.2 and 5.3.3 repeatedly. □

5.4 Epistemic Model Checking

In this section we consider the model checking problem of formulas in $\mathcal{L}_{KU}$ over a SOT model. There exist different off-the-shelf model checkers [157, 116] for the logic of knowledge and time. Traditionally, their main application domains are distributed systems and protocol verification. Section 5.4.1 explores the use of epistemic model checking for software verification by encoding a SOT model and $\mathcal{L}_{KU}$ formula into an MCMAS model. As shown by our experiments, the performance is inversely proportional to the inputs domain size. Section 5.4.2 introduces a new model checking algorithm which is tailored to the verification of noninterference and declassification policies. The algorithm transforms a SOT and a policy formula into an existentially quantified FOL formula which can be checked efficiently by an SMT solver.

5.4.1 Encoding a SOT as an Interpreted System

MCMAS is an epistemic model checker which can be used to model a multiagent system and reason about its epistemic and temporal properties [157]. Any SOT can be encoded into an interpreted system model, similar to Def. 5.2.2, on which MCMAS can be used to prove information flow properties. The encoding simply transforms the SOT in an MCMAS model with perfect recall where the *Environment* agent simulates "internal" executions in the SOT model, while an *Attacker* agent collects the observable traces generated during those SOT executions. An internal variable of the *Environment* agent, *state*, records the current node of the SOT execution. For all SOT node n, the *Environment* agent's protocol can emit an action "go to n" only if *state* corresponds to a predecessor of n and the path condition associated with n holds. The associated evolution function sets *state* to n and assigns the output expression of n to a variable, *out*, observable by the *Attacker* agent. In order to model a perfect recall attacker, the *Attacker* agent possesses a variable for each "depth" level in the SOT, $obsL_i$. At every step s, the *Attacker* agent copies the content of the *out* variable into its $obsL_s$ variable, and updates its state in order to copy next into $obsL_{s+1}$.

Any SOT can be systematically transformed to an interpreted system by following Template 1 where the SOT has n nodes, m inputs, d max depth, where

Template 1 SOT to MCMAS model

Environment agent
  Obsvars: out
  Vars: $\text{in}_1, \ldots, \text{in}_m$, state: $\{\text{init}, \text{s}_1, \ldots, \text{s}_n\}$
  Actions: start, $\text{gos}_1, \ldots, \text{gos}_n$
  Protocol:
    $\vdots$
    Pc_i and state = $\text{s}_{pred(i)}$: $\{\text{gos}_i\}$
    $\vdots$
  Evolution:
    $\vdots$
    out = e_i and state = s_i if $\{\text{gos}_i\}$
    $\vdots$
Attacker agent
  Vars: lev, $\text{obsL}_1, \ldots, \text{obsL}_d$
  Actions: none
  Protocol: none
  Evolution:
    (lev = lev + 1) if lev = 0
    $\vdots$
    (lev = lev + 1) and obsL_l = out if lev = l
    $\vdots$
Initial state
  state = init and lev = 0
Formula
  AG($\bigwedge_{secret,v}$!K(Attacker,!($secret = v$))

$pred(i)$ is a predecessor of node i, Pc_i and e_i are the path condition and output expression associated with node i, *secret* is any secret to be protected and v any value this secret can take. The correctness of such transformation is then stated by the following theorem.

Theorem 5.4.1 (SOT-IS equivalence) *Let SOT be a symbolic output graph and IS the associated interpreted system derived by the previous construction. Then,*

$$\mathcal{M}(SOT) = \mathcal{M}(IS)$$

Performance Analysis A number of experiments have been performed and reported in the last column of Fig. 5.5. The SOT generated for each use case (described in Sect. 5.5) has been encoded as an input to the MCMAS model checker [157] by the transformation presented above. The evaluation results show a strong correlation between the domain size of the input variables and the running time of the model checker. The numbers refer to the running time (in seconds) of MCMAS, where the domain of integer variables is the interval $[-50, 50]$. Most of the mediun-size examples fail even for small domains due to the huge size of the epistemic formula that we verify. Moreover, our experiments show that also for simple formulas the running time increases with the domain size.

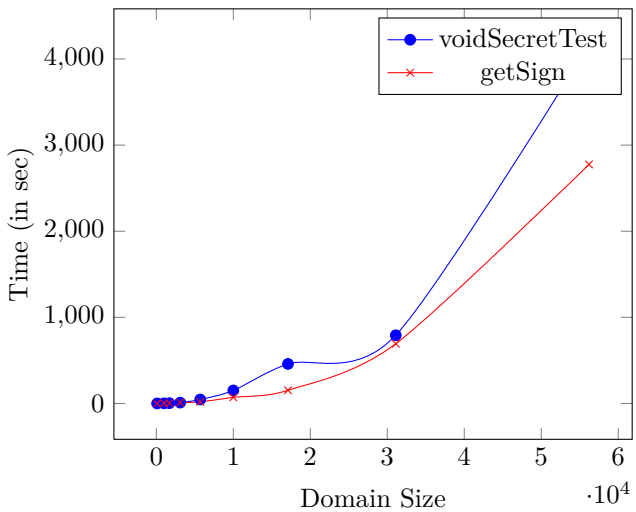


Figure 5.3: Running Time vs Domain Size

The graph in Fig. 5.3 (abscissa in multiple of 10^4) represents the MCMAS running time as a function of the input domain size for two simple examples. In both cases MCMAS verifies a simple epistemic formula which is true in one example (*voidSecretTest*) and is false in the other (*getSign*). Beside the steep increase of

running time with domain size, one can also note that proving a formula which is true in a model requires more time than disproving a similar formula is false in a model of roughly same size.

5.4.2 A New Model Checking Algorithm

It is known that model checking via BDDs works well when the size of the domain is relatively small [67]. In software model checking domain size can be large or even infinite, therefore model checking can be problematic, as confirmed by our experiments. To face this problem, we present a new algorithm that reduces the epistemic model checking over SOT models to SMT solving of a formula which only contains variables in existential form. While in general the transformation to existential form is not possible for every formula, this can be done for the information flow properties we are interested in verifying.
Given a formula ϕ and a model M associated with an SOT S, we define a transformation $T(S, \phi)$ and prove that ϕ holds in $\mathcal{M}$ iff $T(S, \phi)$ is valid. We then derive the noninterference-like formulas which can be verified by an SMT solver.
In what follows $\vec{O}_n$ is the tuple of output expressions encountered on an SOT path, from node Start to node n. We write $\vec{O}_n = \vec{O}_{n'}$ to denote the component-wise equality between tuple expressions and, $N(S)$ to denote the nodes of an SOT S.

Definition 5.4.1 ($T(S, \phi)$) *Given an SOT S and a formula ϕ in $\mathcal{L}_{KU}$, $T(S, \phi)$ is defined as:*

$$T(S, \phi) = \bigwedge_{n \in N(S)} \forall \vec{x}(Pc_n \Rightarrow T(S, n, \phi))$$

where $T(S, n, \phi)$ is defined as

- $T(S, n, b) = b$
- $T(S, n, \neg\phi) = \neg T(S, n, \phi)$
- $T(S, n, \phi_1 \rightarrow \phi_2) = T(S, n, \phi_1) \rightarrow T(S, n, \phi_2)$
- $T(S, n, \forall \vec{u}.\phi) = \forall \vec{u}.T(S, n, \phi)$
- $T(S, n, K\phi) = \bigwedge_{n' \in N(S)} \forall \vec{x}'.\ ([Pc_{n'}]' \Rightarrow$
  $\vec{O}_n = [\vec{O}_{n'}]' \Rightarrow [T(S, n', \phi)]')$

where $[F]' = F[\vec{x} \mapsto \vec{x}']$ is a renaming of all free variables $\vec{x}$ in F with $\vec{x}'$.

The intuition behind the transformation $T(S, \phi)$ is that each node in $N(S)$ represents an epistemic state in which both the path condition and the sequence of output expressions up to that node are true (atomic propositions in Def. 5.2.2). Consequently, if a formula ϕ is weaker, i.e. implied, than the atomic propositions for all nodes, ϕ is true in the SOT model.

Proposition 5.4.1 *Let S be an SOT and $M(S)$ the corresponding model. Then for all formula ϕ*

$$M(S) \models \phi \Leftrightarrow \models T(S, \phi)$$

PROOF. [Proof Sketch] Let Π be a path in S, with *Start* and *End* node removed, and the sequence of pairs $(Pc_1, e_1) \Rightarrow \cdots \Rightarrow (Pc_k, e_k)$ occurring in Π. Then the model $M(S) = \{\pi \mid \exists \Pi \in Paths(S).\ len(\pi) = len(\Pi) \wedge \forall i.\ \sigma(\pi, i) \models Pc_i \wedge \alpha_i = \sigma(\pi, i)(e_i)\}$. ($\Rightarrow$) We show, by structural induction on ϕ, for all $\pi, i \in M(S)$, that if $\pi, i \models \phi$ then $Pc_i \Rightarrow T(S, i, \phi)$ is valid. Suppose $\phi = K\phi'$. By definition of satisfaction, for all $\pi', i' \in M(S)$, if $trace(\pi, i) = trace(\pi', i')$ then $\pi', i' \models \phi'$. We then show $\forall \vec{x}(Pc_i \Rightarrow \bigwedge_{i' \in N(S)} \forall \vec{x}'.\ ([Pc_{i'}]' \Rightarrow \vec{O}_i = [\vec{O}_{i'}]' \Rightarrow [T(S, i', \phi')]'))(**)$ holds, which follows from definition of $M(S)$ and induction hypothesis. Other cases are easy. ($\Leftarrow$) Let ϕ be a formula and assume $T(S, \phi)$ holds. We show that $M(S) \models \phi$. Suppose $\phi = K\phi'$. Then $(**)$ is true. Consider the tuples of values $\vec{c}^*, \vec{O}^*$ such that $Pc(\vec{c}^*)$ and $\vec{O}_i(\vec{c}^*) = \vec{O}^*$ and a state σ^* with identifier values from $\vec{c}^*$. In particular, $\sigma^* \models Pc_i$ and $\sigma^*(\vec{O}_i) = \vec{O}^*$. Again by assumption consider $\vec{c_1}^*$ where $[Pc_{i'}]'(\vec{c}_1^*)$ and $[\vec{O}_{i'}]'(\vec{c}_1^*) = \vec{O}^*$, hence the state σ_1^* mapping identifiers to values $\vec{c}_1^*$ implies $\sigma_1^* \models [Pc_{i'}]'$ and $\sigma^*([\vec{O}_{i'}]') = \vec{O}^*$. By hypothesis and these facts the claim follows. □

We can now safely use transformation T for noninterference-like formulas.

Corollary 5.4.1 *Let S be an SOT associated with program P and AK the noninterference formula. Then, $P(\vec{l}, \vec{h})$, program P with high identifiers $\vec{h}$ and low identifiers $\vec{l}$ is noninterfering iff the following formula is unsatisfiable.*

$$\exists \vec{l}, \vec{h}, \vec{h}'.\ \bigvee_{n \in N(S)} (Pc_n(\vec{l}, \vec{h}) \wedge (\bigwedge_{n' \in N(S)} \neg(Pc_{n'}(\vec{l}, \vec{h}') \ \wedge\ \vec{O}_n(\vec{l}, \vec{h}) = \vec{O}_{n'}(\vec{l}, \vec{h}'))))$$

PROOF. Applying transformation T to the negation of AK, defined in Sect. 5.2.4, and substituting $\vec{l} = \vec{l}'$ and $\vec{h} = \vec{u}$, proves the claim. Indeed, $AK :=$ $\forall \vec{v}, \vec{u}.((\vec{l} = \vec{v}) \Rightarrow L(\vec{l} = \vec{v} \wedge \vec{h} = \vec{u}))$, then $T(S, AK) = \bigwedge_{n \in N(S)} \forall \vec{l}, \vec{h}.\ (Pc_n(\vec{l}, \vec{h}) \Rightarrow T(S, n, AK)) = \bigwedge_{n \in N(S)} \forall \vec{l}, \vec{h}.\ (Pc_n(\vec{l}, \vec{h}) \Rightarrow \forall \vec{v}, \vec{u}.\ (\vec{l} = \vec{v} \Rightarrow \neg \bigwedge_{n' \in N(S)} \forall \vec{l}', \vec{h}'.$ $(Pc_{n'}(\vec{l}', \vec{h}') \Rightarrow O_n(\vec{l}, \vec{h}) = O_{n'}(\vec{l}', \vec{h}') \Rightarrow \neg(\vec{l}' = \vec{v} \wedge \vec{h}' = \vec{u}))))$. Then the negation of the last formula is true if $\bigvee_{n \in N(S)} \exists \vec{l}, \vec{h}.\ (Pc_n(\vec{l}, \vec{h}) \wedge \exists \vec{v}, \vec{u}.\ (\vec{l} = \vec{v} \wedge \bigwedge_{n' \in N(S)} \forall \vec{l}', \vec{h}'.\ \neg(Pc_{n'}(\vec{l}', \vec{h}') \wedge O_n(\vec{l}, \vec{h}) = O_{n'}(\vec{l}', \vec{h}') \wedge (\vec{l}' = \vec{v} \wedge \vec{h}' = \vec{u}))))$ which holds iff the formula is satisfiable for $\vec{l} = \vec{l}'$ and $\vec{h} = \vec{u}$. Finally we perform these substitutions in the formula and derive the claim. □

In case of a declassification policy $\phi(\vec{l}, \vec{h})$ one can similarly apply transformation T and obtain a formula $T(AK) \wedge \phi(\vec{l}, \vec{h})$. We now apply the algorithm in Corollary 5.4.1 to our running example.

Example 5.4.1 *Consider the SOT S in Fig. 5.2 corresponding to the program in Example 5.2.2 which we explained to be noninterfering. This means that the*

following formula must be unsatisfiable.

$$\exists secret, secret'.\ \bigvee_{n \in N(S)} (Pc_n(secret) \wedge (\bigwedge_{n' \in N(S)} \neg(Pc_{n'}(secret') \wedge \vec{O}_n(secret') = \vec{O}_{n'}(secret'))))$$

Consider a node $n \in N(S)$, say the one on top left, where $Pc_1 = (0 < secret \leq 2)$ and $O = 0$. Then the formula is satisfiable if there exists a value of secret where $Pc(secret)$ holds, for instance $secret = 1$, and a value of $secret'$ that falsifies, for all nodes, the path conditions or the equality between output expressions. We only do the check for nodes at the same level of n, otherwise the output sequences will never be equal. Moreover, nodes at the same level have equal outputs, hence the formula can only be falsified (hence the condition satisfied) by a value of $secret'$ that sets to false all path conditions at that level. But since some of the conditions are pairwise disjoint, this will never be the case. Consequently the formula is unsatisfiable for node n. The check for other nodes can be done similarly and prove that P is noninterfering.

5.5 Implementation

The theory presented above has been implemented in a prototype called ENCoVer [39]. For the extraction of the symbolic output tree (SOT) from Java bytecode, ENCoVer relies on *Symbolic PathFinder* (SPF) [145], an extension of *Java PathFinder* [217]. SPF exercises all possible execution paths of the analyzed program by means of concolic testing [146]. During this phase, SPF computes and maintains symbolic expressions representative of the current path condition and of the value of every variable for the current path under test. Whenever a statement rendering a value "observable" is executed, ENCoVer creates a new node in the SOT under generation using the symbolic expressions corresponding to this observable value and the current path condition. After this first phase corresponding to the SOT generation, ENCoVer converts the SOT into an interference formula (f) with free variables. This formula, with its free variables existentially quantified, is the negation of the noninterference formula applied to the program analyzed, as described in Section 5.4. Any assignment to the free variables that renders the formula f true is a counterexample proving that the program is not noninterfering. Finally, ENCoVer feeds the formula f to a satisfiability modulo theory (SMT) solver (Z3 [98] in the current implementation). If the SMT solver answers that the formula is unsatisfiable, then the analyzed program is deemed noninterfering. Otherwise the program is declared interfering, and the assignment provided by the SMT solver is returned as a counterexample of the noninterference behavior of the analyzed program.

ENCoVer has been implemented in Java as an extension of *Java PathFinder* (JPF). The extension by itself has 86 classes/interfaces and 6 KLOC as computed

by CLOC [95], and 161 KLOC including the required parts of SPF. The class of programs that the current implementation of ENCoVer can handle is indirectly limited by the class of programs SPF (JPF core and its symbc extension) can handle and the class of expressions Z3 can solve. There is no intrinsic limitation induced by the specifics of ENCoVer itself. Theoretically SPF can execute any Java bytecode, however in practice SPF is limited by missing implementations for some native libraries (such as java.io and java.net), a few bugs (such as NullPointer exceptions being reported as NoSuchMethod exceptions), and of course state space explosion (particularly when dealing with multithreaded programs with loose synchronization constraints). In the current implementation (due to the way SPF handles booleans, and differences between SPF expressions and Z3 expressions that requires typing in order to translate from one to the other), ENCoVer is limited to the manipulation of integer expressions as described by the Core and Ints theories of the SMT-LIB standard [48]. Z3 can solve a fair number of formulas based on those expressions [65, 47]. In the future, the class of programs handled by ENCoVer should grow due to continuous development on SPF and Z3.

5.5.1 Case study

As a main case study, ENCoVer has been applied to the security-oriented case study of the HATS project [129]. This case study, Tax Record (TR), simulates the interactions between a server handling tax records, tax payers, tax checker entities, and a charity. Tax payers can dedicate part of their payments to a charity. To every tax payer is associated a tax record which is initialized with her incomes, and to every tax record is associated a tax checker. The tax payer can query the amount of taxes due, and perform a payment indicating how much is to be given to the charity. After each payment, the associated tax checker verifies that the cumulated payments cover the sum of the taxes due and the charity donation. If that is the case, the tax record is frozen and no further modification can be made. Once all the tax records have been frozen, the server informs the charity of the sum of money given by the tax payers.

The Java implementation has 8 classes/interfaces (as shown in Fig 5.4) and 267 LOC. There is one class for each of the two "types of object" (TaxServer and TaxRecord, ranged over by O) and each of three "types of principal" (TaxPayer, TaxChecker and Charity, ranged over by P). The three interfaces (TaxServer4charity, TaxRecord4taxPayer and TaxRecord4taxChecker, ranged over by $O4P$) describe the actions/queries that principals of type P can perform on objects of type O. The implementations of TaxPayer, TaxChecker and Charity describe the intended processes those principals should follow. However, "bad" principals of type P could perform different actions on objects of type O, but only using methods listed in interface $O4P$ and implemented in O. Two taxation schemes have been implemented. The tax rate is either fixed ($F\%$) and computed by a simple multiplication, or variable over "slices" of income and computed in a while loop by cumulating the taxes for each slice of the income where the n^{th} slice of 10 K$ is taxed $(n \times V)\%$.

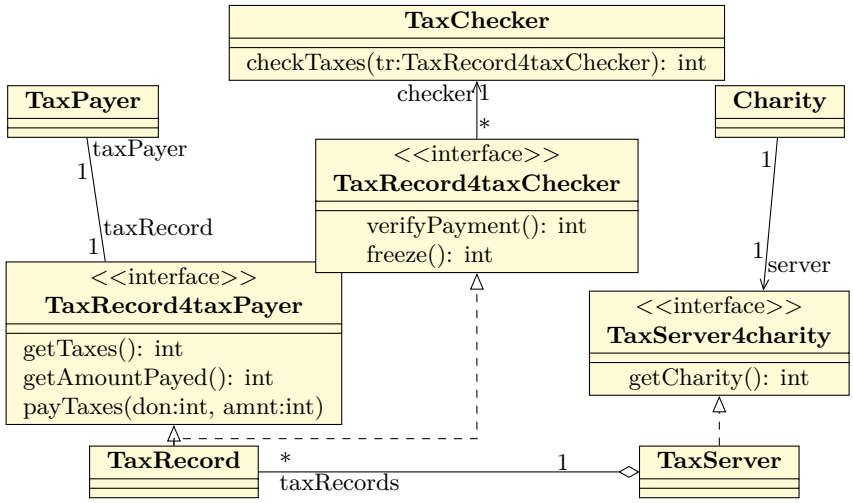


Figure 5.4: Class diagram of the Tax Record case study

From a security point of view, one property to verify is whether a given tax payer is able to deduce any information about the income, payments and donation of other tax payers by triggering and observing the result of actions specified in TaxRecord4taxPayer. Similarly, the tax checker is only allowed to know if the cumulated payments are equal to or higher than the sum of the taxes and donation of a tax record, and, if that is the case, to know the amount of overpayment. Finally, the charity should not be able to learn anything except the cumulated amount of donations.

5.5.2 Application of ENCoVer to the TR case study

The HATS' case study is intrinsically an interactive program whose behavior mainly depends on the actions of the tax payers. In order to extract the SOT from the program, Symbolic PathFinder (SPF), which relies on a concolic testing approach [185], executes the program to be verified. This requires to provide an additional executable program simulating the behavior of the different participants involved in an execution of this interactive program. Three different scenarios have been examined. The first scenario (`smpl`), involves a single tax payer (Alice) which queries for her amount of taxes and pays that exact amount without making any donation. The only input in this scenario is the income of Alice. The second scenario (`oneP`) involves the same tax payer initially performing a first payment and donation, then, if she has under-payed, queries for her amount of taxes and pay what remains, including the donation. The inputs are Alice's income, donation and first payment. It is to be noted that donation can be zero, which is equivalent to not making a donation. The last scenario (`twoP`) involves two tax payers, Alice

and Others, representing all the other tax payers. Both act as Alice in the second scenario. There are 6 inputs: incomes, donations and first payments of Alice and Others.

For every scenario and taxation scheme, ENCoVer is used multiple times to verify the noninterfering behavior of the program with regard to the 3 different principals (Alice, tax checker and charity, ranged over by P) under different policies regarding values that have to be protected from those principals. Each analysis involves a different configuration of ENCoVer. Among other parameters such as input domains, there are 3 main parameters to configure: the input values (or expressions) known by P at the beginning (the low values in the theory), the input expressions that should be kept secret from P (the high values), and finally the events and associated values that are observable by P. This last parameter is configured by providing an expression with wild-cards specifying which method calls are observable by P and which parameter or return value P will observe. In the case of the tax checker, resp. charity, the configuration of this parameter indicates that the return value of any method in TaxRecord4taxChecker, resp. TaxServer4charity, is observable. In the case of Alice, specifying that the return value of any method in TaxRecord4taxPayer is observable would not allow ENCoVer from distinguishing between observations made by Alice and Others. Therefore, the SOT would contain observations made by both, instead of the observations made by Alice only. However, the expression specifying observable events may include some runtime values of method call parameters. To specify the events observable by Alice, a method `obs(String, int)`, taking as parameter a tax payer name and another value, is coded with an empty body. The observable expression is set to `*.obs("Alice", 0)` and, in any method m specified in TaxRecord4taxPayer, a call to this `obs` method is inserted with parameters the name of the tax payer for this tax record and the value to be returned by m (`obs(this.taxpayerName, res)`).

Figure 5.5 contains the evaluation results. The remainder of this section focuses on the noninterference analysis results for the Tax Record case study in column 4 (ENCoVer:NI) of Figure 5.5. The relevant tests are named S-P-R, where S indicates the scenario, P is the principal for which the program is verified, and finally R specifies taxation scheme, `Fixed` or `Variable`. For the `smpl` scenario, all configurations are found noninterfering. The only input is the income of Alice, which is known by Alice and has no relation to the values observed by charity (0, as there is no donation in this scenario) and taxChecker (0, as Alice pays directly the exact amount of taxes due). For the `oneP` scenario, the inputs are the income, donation and first payment of Alice known by Alice and hidden from charity and taxChecker. Obviously, this scenario is noninterfering from Alice's point of view, but not from the point of view of charity as the only donation is Alice's. For the principal taxChecker, many different configurations have been tested: In the taxChecker1 case, the declassification policy is "$income \times F\% + donation > payment$", and for taxChecker2 it is "$income \times F\% + donation - payment$". ENCoVer finds the configuration interfering for taxChecker1 and noninterfering for taxChecker2, as expected. Indeed, the value declassified in the taxChecker1 case, resp. tax-

Checker2 case, is a lower bound, resp. upper bound, of the value revealed to the tax checker in the fixed tax rate variant. The exact value revealed to taxChecker in the specification of TaxRecord is "if $income \times F\% + donation > payment$ then -1 else $payment - (income \times F\% + donation)$". The configuration taxChecker3 corresponds exactly to the declassification of this formula. For the variable tax rate case, the expression computing the taxes $((\sum_{n=1}^{N} n \times V\% \times slice) + ((N+1) \times V\% \times (income \bmod slice))$ where the n^{th} slice is taxed $(n \times V)\%$ and $N = income \div slice$ is the number of full slices) can be declassified to the taxChecker by rewriting $\sum_{n=1}^{N} n$ as $((N+1) \times N/2)$. This declassification corresponds to the configuration taxChecker4. The case of the `twoP` scenario, is similar to the previous case for Alice and taxChecker. However, this time there are two different donations, one from Alice and one from Others. By declassifying "*donationAlice* + *donationOthers*" to charity, ENCoVer concludes that charity does not learn more than is allowed. In conclusion, apart from potential efficiency problems that are addressed in the next section, the ENCoVer prototype behaves as expected and can handle the majority of configurations of the tax record scenarios.

5.6 Evaluation

ENCoVer has been used to verify multiple test programs. Figure 5.5 contains data for some of the tests. The first test program, `empty`, is used as a base reference for normalizing the number of instructions executed by JPF. The two tests `getSign` and `voidSecretTest` are used to verify the correctness of the answer returned by ENCoVer. The program `getSign` takes a secret h as input and returns -1, resp. 0 or 1, if h is negative, resp. zero or strictly positive. This program is obviously interfering. The program `voidSecretTest` tests if its secret input h is equal to 0, and returns h if it is true, 0 otherwise. As this program always returns 0, it is noninterfering.

The "double while" running example used previously corresponds to the tests named `whileLoops-`X, where X is the maximum number of loops (2 in the case of the running example). The same specification (2 consecutive iterative structures whose total number of iterations is X) has been implemented using recursive method calls instead of while statements. However, as the results are similar to the double while implementation, they are not reported in Fig. 5.5. The other lines correspond to different configurations for the use case described in the previous section.

5.6.1 Efficiency

Two test cases caused the ENCoVer tool to fail completely: `twoP-charity-V` and `twoP-taxChecker4-V`. The analysis of the logs reveals that ENCoVer runs out of memory while generating the interference formula, consisting of a large number of identical subformula objects. We believe this problem can be remedied by

TEST	JPF		ENCoVer										
	States	Inst	NI	Timing (in ms)				SOT			Fml			MC
				O (in s)	E	G	S	N	D	W	V	A	I	
empty	1	0	Y	.4	9	3	4	0	0	0	0	0	0	.0
getSign	13	48	N	.6	115	2	36	3	1	3	2	34	68	.1
voidSecretTest	3	18	Y	.5	88	2	18	2	1	2	2	11	22	.1
whileLoops-2	23	181	Y	.6	137	7	53	9	2	5	2	219	454	.2
whileLoops-30	1059	6873	Y	143.0	1795	8980	131662	555	30	33	2	579371	4150614	488
whileLoops-40	1809	11543	?	2595.2	2415	48867	-	940	40	43	2	1680161	15359218	-
smpl-Alice-F	5	877	Y	.6	173	3	8	3	3	1	2	17	62	.4
smpl-Alice-V	1067	19382	Y	3.3	1528	249	1054	63	3	21	2	18777	103926	2.3
smpl-charity-F	5	877	Y	.6	179	3	8	1	1	1	2	8	26	.9
smpl-charity-V	1067	19382	Y	2.5	1470	86	576	21	1	21	2	5968	31098	.1
smpl-taxChecker-F	5	877	Y	.6	167	3	8	1	1	1	2	8	36	.1
smpl-taxChecker-V	1067	19382	Y	2.7	1452	88	724	21	1	21	2	5968	37650	.1
oneP-Alice-F	13	1353	Y	2.3	1900	6	12	5	4	2	6	42	236	14.4
oneP-Alice-V	2185	32604	Y	6.9	3659	240	2546	87	4	24	6	29114	179100	-
oneP-charity-F	13	1353	N	2.3	1861	4	42	2	1	2	6	28	107	-
oneP-charity-V	2185	32604	N	4.7	3517	96	637	24	1	24	6	7916	42957	-
oneP-taxChecker1-F	13	1353	N	2.3	1872	4	27	3	2	2	6	32	154	-
oneP-taxChecker2-F	13	1353	Y	2.4	1895	4	25	3	2	2	6	32	154	-
oneP-taxChecker3-F	13	1353	Y	2.3	1844	4	24	3	2	2	6	32	164	-
oneP-taxChecker4-V	2185	32604	Y	128.9	3632	129	124709	45	2	24	6	14500	84462	-
twoP-Alice-F	37	3578	Y	6.3	5820	6	26	5	4	2	12	57	266	-
twoP-Alice-V	54601	824013	?	2541.3	2537857	293	-	87	4	24	12	29129	179130	-
twoP-charity-F	37	3578	Y	6.5	5962	10	45	4	1	4	12	107	588	-
twoP-charity-V	-	-	?	-	-	-	-	-	-	-	-	-	-	-
twoP-taxChecker3-F	37	3578	Y	6.6	5852	12	250	9	4	3	12	159	1134	-
twoP-taxChecker4-V	-	-	?	-	-	-	-	-	-	-	-	-	-	-

- JPF
  - States: number of states encountered during concolic execution
  - Inst: total number of instructions executed (normalized such that the value for the empty test is 0 rather than 2926)
- ENCoVer
  - NI: Y iff ENCoVer concludes that the program is noninterfering
  - Timing: given in ms (O: overall in s; E: model extraction (JPF+symbc); G: interference formula generation; S: interference formula satisfiability checking)
  - SOT: information related to the SOT (N: number of nodes; D: depth of the SOT (correspond to the longest possible sequence of outputs); W: width of the SOT (corresponds to the maximum number of nodes at any level))
  - Fml: information related to the interference formula (V: number of distinct variables; A: number of atomic formulas; I: number of instances of variables or constants)
  - MC: timing in s for MCMAS model checker (independent additional execution on the generated model; not included in the overall time taken by ENCoVer when using SMT resolution)

Figure 5.5: Evaluation results

subformula sharing. As a side effect, once the interference formula is composed of references to a smaller number of unique subformulas, it will be possible to feed it in incremental steps to Z3. It is expected that this will allow Z3 to handle cases where it runs out of memory while trying to satisfy the interference formula. This is indeed what prevents Z3 to conclude for the test cases `whileLoops-40` and `twoP-Alice-V`.

The test case `whileLoops-30` shows that ENCoVer can handle programs with nontrivial SOT's. Symbolic PathFinder (SPF) extracted more than 500 different SOT nodes. A single execution of `whileLoops-30` outputs 30 different values, for which there exists 33 different potential output expressions depending on the path followed for at least one of those values. As suggested by the tax record use case, many "real" programs are likely to produce smaller SOT's with less diverse output expressions. It is noteworthy that for `whileLoops-30`, Z3 needs only a little more than 2 minutes to conclude that the interference formula is unsatisfiable.

The results for the tax record study show that the extraction of the output behavioral model can be quite time consuming especially when the number of paths explodes, mainly due to while loops.

ENCoVer's memory handling can be improved. However, the results demonstrate that the approach proposed in this paper can be used to verify complex information flow policies on non-trivial programs with complex, control-dependent information flow.

5.7 Related Work

The most closely related work is that of Cerny and Alur [71] which presents an automated analysis of conditional confidentiality for Java midlet methods. A property f is conditionally confidential (CC) wrt. to property g if for every execution r for which property g holds another execution r' exists with the same observation as r but such that r and r' disagree on f. This condition is expressed as a formula over program identifiers involving existential and universal quantifiers. To check the formula over- and under- approximations of reachable states are computed for every program location and universal quantification is carefully set to take place over a bounded domain. A tool called CONAN is developed for analyzing CC of Java midlet methods. We strongly believe that CC can be expressed in epistemic logic by the formula $(g \Rightarrow (Lf \wedge L\neg f))$, where, intuitively, g is a property known by the observer and f is the property to protect. In our case the corresponding formulas will involve existential quantifiers only and they can be immediately fed to an SMT solver. Moreover, the noninterference-like properties we are verifying are much stronger than CC, and we expect to handle the weaker properties as well. On the tools side, ENCoVer performs global analysis for Java programs and is fully automatic. It would be interesting to further investigate how an extension of the epistemic logic considered here relates to $CTL \approx$, which can express CC [19] properties.

Halpern and O'Neill [128] introduce a framework for reasoning about secrecy requirements in multiagent systems. They show how the interpreted systems formalism [112] can be used to express in a clean way different trace-based information flow properties both for synchronous and asynchronous systems. Nondeterminism and probability are also considered. The definition of secrecy is based on an abstract model, the run-and-systems model, which is different from the primary concern of this paper, language-based security. Moreover, they do not consider the verification problem. Another security notion, related to secrecy, is that of opacity [66, 107], which models the ability of a system to keep some critical information secret. The verification techniques presented in this paper can also be applied to opacity. Askarov and Sabelfeld introduce the gradual release model [28, 29] where attackers knowledge is modeled as equivalence relations on input states. A verification technique based on security type systems and monitors is used to verify gradual release for a while language with inputs and and outputs. Other language-based approaches have been used to characterize the attackers power or the declassified information, by means of partial equivalence relations [201] or abstract interpretations [119]. We believe [38] that our epistemic framework can nicely capture these approaches and move a step closer to their verification.

5.8 Conclusion

In this paper we have considered the verification problem for noninterference and declassification policies expressed as formulas in epistemic logic. We have used concolic testing (a mix of concrete and symbolic execution) to obtain an abstract model of the original program such that the verification problem for the epistemic logic is brought within scope of current SMT solvers. This is done by reducing the problem of verification of noninterference and declassification into the satisfiability of a formula that contains variables in existential form only. As showed by the case studies our approach is quite elegant and able to handle tricky cases of information flow, even for programs of non-trivial size. The ENCoVer prototype performs a precise sensitive global analysis and relies on a clear separation between security policy and program text. ENCoVer indicates that recent advances in SMT solving can be combined with symbolic techniques to reduce false alarms and scale up to real software for the case of information flow analysis. Moreover we have showed how to transform the model generated by concolic testing as an interpreted system, which can be subsequently used to for epistemic model checking.

Limitations and Future Work Many limitations of the approach we put forward are due to constraints imposed by the tools used for implementation. On the other hand, the class of programs we can certify automatically is still of interest, as shown by the experiments.

Assuming that inputs are read at the start of program execution rules out a class of reactive programs that receive inputs during the execution [57, 182]. One

way to overcome this restriction is to rewrite the original program to an equivalent one that reads all inputs prior to execution start and uses them as needed. This can be done for the class of interactive deterministic programs [80]. In particular, one can rewrite the original program by replacing internal inputs with a dummy output operation and introducing a fresh variable which is read in the beginning of execution. A more general account of interactive programs must take attacker strategies into account [182].

Another limitation is that our tool only supports a bounded model of runtime behavior. Automatic invariant generation techniques may be integrated with ENCoVer to speed up the analysis and overcome this limitation.

A further issue concerns the background arithmetic theories that the SMT solver is able to handle. Currently Z3 works well with linear arithmetics, while non linear constraints are not handled [98]. Consequently, it becomes crucial to apply abstraction techniques, e.g. predicate abstraction [34], when the path conditions represent as non-linear constraints. Moreover performing modular verification at level of Java methods, would improve performance at cost of losing the precision that global analysis provides. We plan to address these techniques in the future.

Acknowledgements. The authors would like to thank the anonymous reviewers, as well as the participants to the Åre workshop, for their helpful comments. This work was partially supported by the EU-funded FP7-project HATS (grant № 231620).

Chapter 6

Automating Information Flow Analysis of Low Level Code

Musard Balliu and Mads Dam and Roberto Guanciale

Abstract

Low level code is challenging: It lacks structure, it uses jumps and symbolic addresses, the control flow is often highly optimized, and registers and memory locations may be reused in ways that make typing extremely challenging. Information flow properties create additional complications: They are hyperproperties relating multiple executions, and the possibility of interrupts and concurrency, and use of devices and features like memory-mapped I/O requires a departure from the usual initial-state final-state account of noninterference. In this work we propose a novel approach to relational verification for machine code. Verification goals are expressed as equivalence of traces decorated with observation points. Relational verification conditions are propagated between observation points using symbolic execution, and discharged using first-order reasoning. We have implemented an automated tool that integrates with SMT solvers to automate the verification task. The tool transforms ARMv7 binaries into an intermediate, architecture-independent format using the BAP toolset by means of a verified translator. We demonstrate the capabilities of the tool on a separation kernel system call handler, which mixes hand-written assembly with gcc-optimized output, a UART device driver and a crypto service modular exponentiation routine.

6.1 Introduction

The ultimate goal of information flow analysis is to establish confidentiality and integrity properties of real code executing on commodity CPUs. In the literature,

normally this problem is addressed at the source code level. There it may be more forgiving to ignore messy low level problems, e.g. regarding timing, complex control flow, or hardware specifics. Also, one may appeal to special compilers that avoid difficult optimizations, or work around machine features such as caching, instruction reordering, concurrency, I/O, interrupts, bus contention and so on, that are difficult to handle in a precise manner.

Sometimes, however, source level analysis is less suitable. This is certainly the case when dealing with third-party code, but it applies in other cases too, for instance, for heavily optimized or obfuscated code, and for kernel handler routines that manipulate security sensitive peripherals such as privileged processor registers, MMUs, and bus and interrupt controllers.

The literature has two "standard" approaches to information flow control (IFC) for low level languages: (a) For static verification, most authors, cf. [168, 53], have attempted to reimpose typing and high level structure at the assembly or byte code level, in order to reuse standard type-based techniques for high level languages. For instance, [168], uses this approach for typed assembly language, and [53] takes a related approach to Java bytecode. (b) Most work, however, has focused on dynamic techniques, often using some combination with static analysis to generate labels, or tags, to help minimize the dynamic overhead, cf. [61, 137, 210, 153, 32]. For instance, [32] proposes a machine architecture with hardware-supported tag propagation to support dynamic information flow tracking.

Neither of these schools are very helpful, though, when it comes to the problem we have set out to study: Information flow analysis for low level code on commodity processors. In this domain, existing static approaches are too imprecise due to lightweight (data/flow/path/timing-insensitive) analysis, while dynamic approaches suffer from the well known problem of label creep and introduce undesired runtime overhead [197]. Security testing-like techniques [180, 31], which we discuss later, provide impressive results in terms of scalability, however, they are in general unsound and can not directly be used for full verification.

Instead we propose to directly verify relational (i.e. information flow) properties at machine code level, leveraging as much as possible recent progress on low level code analysis tools such as BAP [62], McVeto [213], Vine/BitBlaze [208]. Code for our target machine, ARMv7, is first lifted to a machine-independent intermediary form, BIL, using the BAP tool [62]. This process uses a lifter that is produced from the Cambridge HOL4 model of ARMv7 [135]. This allows the reuse and extension of BAPs program verification back end to symbolically execute the resulting BIL code. We use this to first perform *unary* analysis and then verify relational properties by propagating relational preconditions through each of a pair of related programs until a pair of observation points are reached, that need to be matched, in order for the relational property to hold. These observation points are memory write events, to locations that are statically determined to be observable by some external agent, because of multithreading, or memory-mapped I/O, or for some other reason. Matching is done by SMT solving using STP [118], on formulas that tend to grow huge, but generally rely only on linear arithmetic, uninterpreted functions, and

arrays, and so are not too costly to check. Special care is needed for memory accesses which introduce quantifier alternation, hence we propose an instantiation technique which ensures the resulting formulas are quantifier free.

Three distinguishing features make our information flow analysis both useful and challenging: *loop invariants*, *timing* and *traces*. Loops are handled using (relational) invariants/widening. We point out that relational invariants can be significantly simpler than state invariants as they may not require proving functional correctness of the loop. Our case studies show that the invariants we provide are conjunctions of linear equalities, which, as shown in recent work [206], can be generated automatically. Timing is particularly critical. The timing information is included in the symbolic state and propagated with the other constraints. The model used here scales to functional cost models, i.e. models where the timing cost can be calculated as function of the input instruction, independent of the history. This is evidently realistic only for simple processor architectures such as ARM Cortex-M (but we note that a vast number of such processors are in use today in critical control applications). Richer and tractable timing models that can take into account also features like caches and instruction pipelines are, however, currently not available at ISA level, and we leave this for future work. Finally, the trace-based analysis broadens the number of target applications handled by our technique, including preemptive environments and scheduling.

We are the first to admit that the approach will suffer from scalability problems, for instance due to path explosion, and due to the generally complex and detailed machine state. However, our primary application is separation kernel handler verification, and this domain is generally characterized by critical machine code fragments that are rather small (generally under 1K instructions per handler), but also tricky. The case studies reported in this paper are based on syscall handlers and device drivers of slightly more than 250 lines of ARMv7, produced by a mix of hand-crafted assembly and GCC-optimized C.

Overall, this paper makes both theoretical and practical contributions. On the theoretical side, we present a novel approach for formal relational machine code verification, with focus on information flow security properties. The combination of unary and relational analysis makes our approach appealing for precise security analysis of machine code. We provide a new angle with the inclusion of timing information into the state and with the invariant handling which ensures a nice compositional property over traces. On the practical side, we present the first automated toolset for information flow analysis of ARMv7 binaries. We exercise the tool on non-trivial case studies including separation kernel syscalls, device drivers and crypto routines. For a more thorough discussion of related work, please refer to Sect. 6.8.

6.2 Threat Model and Security

In our target applications, trusted and untrusted agents share and control parts of the system memory. Our goal is to ensure that the only information channels connecting agents to each other are the intended ones. These intended channels can be shared buffers, network connections, or specific communication devices such as the message sending syscall handler considered later in the case study. They can also be memory-mapped devices connecting agents to the external world such as a UART device. In the special case where channels form the usual security lattice, the goal reduces to classical Goguen-Meseguer information flow [122], which requires the state of the untrusted program be unaffected by the state of the trusted one.

We use the ARMv7 program in Fig. 6.1 to elucidate our threat model. The program loads a memory pointer from the address 2048 into the register `R3`. Subsequently, the memory referenced by the pointer is updated three times. The program always terminates with the following effect on the system state: (i) the memory pointer is loaded into the register `R3`, (ii) the registers `R1` and `R2` are updated to zero (lines 0x110 and 0x118), (iii) the "zero flag" `Z` is enabled (the instruction at line 0x110 contains the `S` suffix, thus overriding `Z` according to the result of the executed arithmetic operation [23]) and (iv) zero is written (line 0x114) into the memory referenced by the pointer.

```
0x0f4 MOV R2, #0
0x0f8 LDR R3, [PC+#0x700] //2048
0x0fc STR R2, [R3]
0x100 LDR R1, [PC+#0x2f8] //1024
0x104 ADDS R1, R1, R2
0x108 MOVEQ R2, #1
0x10c STR R2, [R3]
0x110 MOVS R2, #0
0x114 STR R2, [R3]
0x118 MOV R1, #0
```

Figure 6.1: ARMv7 Program

Assume that the system memory from address 0 to 2047 contains the state of a trusted agent and that the remaining part of the memory contains the state of an untrusted agent. If an observing agent is not able to access its own memory while the program is executed, then the above program can be considered secure, since after termination the state of the untrusted agent is unaffected by the state of the trusted one. However, in several scenarios this requirement is not satisfied: (i) the observer controls a device that is mapped to a memory area that belongs to the untrusted agent, (ii) the code is interrupted and scheduled in a preemptive environment, (iii) memory stores have side effects, or (iv) the code is executed in a multi-core setting.

In all these cases, a departure from the usual initial-state final-state account of noninterference is required. In particular, all updates to the untrusted memory (e.g. lines 0x0fc, 0x10c and 0x114) can be monitored by the attacker and reveal secret information. Hence our example program can not be considered secure. In fact, depending on the content of the memory of the trusted agent, the assembly fragment can have the following executions:

1) If the memory at address 1024 is zero (line 0x100) then (i) the instruction 0x104 enables the flag `Z`, (ii) the instruction 0x108 updates the register `R2` to 1, (iii) thus the address referenced by the pointer is updated three times, with the values 0, 1 and 0, respectively.

2) Otherwise (i) the instruction 0x104 disables the flag `Z`, (ii) the instruction 0x108 has no effect, (iii) thus the memory referenced by the pointer is updated three times, always with the value zero.

Consequently, an attacker capable of observing the relevant memory state will in one case see the referenced memory location flicker, and in another case not. The security condition must prevent this phenomenon.

In this paper we work with *observational determinism* [166], defined as follows. Assume a set of configurations C and a transition relation $\rightarrow \subseteq C \times C$. An *execution* is a maximal finite or infinite sequence

$$\pi = C_0 \rightarrow \cdots \rightarrow C_n \rightarrow \cdots \tag{6.1}$$

of configurations related by the transition relation. The initial configuration of π in (6.1) is C_0. A transition may give rise to an observation *obs*. In our case, observations are timed writes to observer readable memory. An *observation trace*, or just *trace*, $trc(\pi)$, of π extracts from π the sequence of observations produced by π. The traces π_1 and π_2 are then *trace equivalent*, if $trc(\pi_1) = trc(\pi_2)$.

Since we assume a concept of observer readable memory, it makes sense to define the relation $C \equiv C'$ by requiring that the observer readable memory of C and C' are the same. This is the familiar notion of observational, or low configuration (state) equivalence. We can then proceed to define observational determinism.

Definition 6.2.1 (Observational Determinism) *A set P of executions π is observational deterministic, if for any pair of executions $\pi_1, \pi_2 \in P$ with initial configurations C_1 and C_2, respectively, if $C_1 \equiv C_2$ then π_1 and π_2 are trace equivalent.*

Observational determinism works well for a class of nondeterministic programs, and it is preserved under refinement [166]. In particular it avoids the quantifier alternation in bisimulation-oriented unwinding conditions. To be accurate, observational determinism presupposes that all nondeterminism can be relegated to the initial state. This is true in our case. However, observational determinism is less suitable when alternation is essential, for instance in the case of strategic reasoning.

6.3 Machine Model

For the theory development we consider deterministic programs written in a simple machine language (SiML) with the following instruction syntax

$$\begin{array}{lcl} \iota & ::= & \mathit{reg} := \mathit{exp} \mid \mathbf{cjmp}(\mathit{exp}, \mathit{exp}, \mathit{exp}) \mid \mathbf{assert}(\mathit{exp}) \\ & & \mid \mathbf{assume}(\mathit{exp}) \mid \mathbf{store}(\mathit{exp}, \mathit{exp}) \mid \mathbf{halt} \\ \mathit{exp} & ::= & \mathbf{load}(\mathit{exp}) \mid \mathbf{bop}(\mathit{exp}, \mathit{exp}) \mid \mathit{reg} \mid \mathit{PC} \\ & & \mid \mathbf{uop}(\mathit{exp}) \mid v \end{array}$$

There are registers $\mathit{reg} \in \mathit{Reg}$, the program counter PC and values $v_{\in} \mathit{Val}$ that are, for simplicity, taken as primitive. Instructions include register assignments, memory stores, conditional jumps, assertions, assumptions, and halt. Expressions include unary and binary operations on constants, register lookups, and memory loads.

A SiML program is evaluated in the context of a register state $\Delta : \mathit{Reg} \mapsto \mathit{Val}$, a program counter $\mathit{pc} \in \mathit{Val}$ and a memory state $\mu : \mathit{Val} \mapsto \mathit{Val}$. We assume a SiML programs be non-self modifying, thus instructions are stored in a separate instruction memory $\Pi : \mathit{Val} \mapsto \iota$ which is usually addressed via the program counter. The instruction memory is a total function and we assume that for each possible address v outside the executable part of the memory $\Pi(v) = \mathbf{assert}(0)$. A *configuration* C is either a tuple $(\Pi, \Delta, \mu, \mathit{pc}, t)$ of instruction memory, register state, data memory, program counter and current execution time $t \in \mathbb{N}$, or an error configuration $\perp$, used to handle failing asserts. The transition relation has the shape $C \rightarrow C'$ where $C \neq \perp$. SiML is a subset of the BAP Intermediate Language (BIL), which is used for lifting the ARMv7 binaries. We refer to [63] for a complete definition of the operational semantics of BIL. Instructions and expressions are evaluated in the context of a configuration. For instance, the assignment $\mathit{reg} := \mathit{exp}$ assigns to register *reg* the value of *exp*, the conditional jump $\mathbf{cjmp}(e_1, e_2, e_3)$ transfers control to e_2 if e_1 is true (non-zero), otherwise to e_3. The $\mathbf{assert}(b)$ statement terminates the program abnormally if b is false, otherwise it has no effect on the state, the $\mathbf{store}(e_1, e_2)$ stores the value of e_2 at memory location e_1 and the expression $\mathbf{load}(e_1)$ loads the value at location e_1. We distinguish between normal and abnormal executions. A *normal* execution, if it ever terminates, executes **halt** as the last instruction. Otherwise the execution is *abnormal* and terminates in the error configuration $\perp$. The length of π is $\mathit{len}(\pi)$, the i-th configuration of π is $\pi(i)$, $\iota(\pi, i) = \Pi(\mathit{pc}_i)$, $\mathit{pc}(\pi, i) = \mathit{pc}_i$, and $t(\pi, i) = t_i$. A *model* $\mathcal{M}$ consists of the set of executions π induced by some set of initial configurations C_0.

Our target applications require reasoning about the execution time of the program. The timing behavior is highly architecture-dependent and is in general very difficult to capture accurately. In this paper we work with a functional time model $\tau : \iota \rightarrow \mathbb{N}$ which assigns a fixed parameter-dependent cost to each instruction. That is if we have $(\Pi, \Delta, \mu, \mathit{pc}, t) \rightarrow (\Pi, \Delta', \mu', \mathit{pc}', t')$ then $t' = t + \tau(\Pi(\mathit{pc}))$. For instance, the execution time of a load instruction will depend on the number of

bytes to load from the memory. Under these assumptions, the execution time is history-independent and the timing model is deterministic.

6.4 Unary Symbolic Analysis

We reason about the behavior of a program by means of forward symbolic analysis. The analysis allows us to build a logical formula, which corresponds to multiple program executions, and leverage first-order reasoning to statically prove program properties. The program is executed on symbolic inputs and, consequently, the state is also symbolic. Initially, registers are mapped to fresh variables, the memory is a variable representing an uninterpreted function and the program counter is a constant. We use exp^s and e^s to range over symbolic expressions, which are built over these initial variables and constants using the standard machinery and have either type memory or type value. In particular, if exp^s is a memory expression and exp_1^s and exp_2^s are expressions of type value, then $exp^s(exp_1^s)$ is an expression of type value representing the lookup of exp_1^s in exp^s, and $exp^s[exp_1^s \mapsto exp_2^s]$ is a memory expression representing the corresponding update.

A symbolic state is a tuple $\Sigma^s = (\Delta^s, \mu^s, pc)$, where Δ^s maps registers to symbolic expressions, μ^s is a symbolic memory expression and pc is the concrete value representing the program counter. A symbolic configuration C^s is a tuple $(\Pi, \Delta^s, \mu^s, \phi, pc, t)$, which extends a symbolic state with a path predicate ϕ, the instruction memory Π and the execution time t. The path predicate ϕ, also path condition, is a symbolic boolean expression built over the initial variables and constrains the set of concrete initial states that execute the path. Usually, the path condition of the initial configuration entails the program preconditions.

Forward symbolic semantics is given by the transition rules on symbolic configurations depicted in Fig. 6.2. Here, we use $\Delta^s, \mu^s \vdash exp \Downarrow exp^s$ to represent the symbolic evaluation of an expression exp in the context (Δ^s, μ^s). For instance, if $\Delta^s, \mu^s \vdash exp \Downarrow exp^s$ then $\Delta^s, \mu^s \vdash \mathbf{load}(exp) \Downarrow \mu^s(exp^s)$. Notice that, since we assume non-self modifying code, we omit the constant instruction memory Π from the rules and the time is increased independently of the processor state. The **cjmp** rules evaluate the jump target in the current symbolic state and then update the program counter and the path condition depending on whether the jump condition is satisfiable. The jump target can be a symbolic expression which requires to resolve all possible targets in the current context. This can be addressed by enumerating all concrete jump targets that are consistent with the path condition, for example using a decision procedure that returns all satisfying assignments of the formula ϕ' in state Σ^s. Another complication arises when considering memory load and store operations. Memory addresses can be symbolic as reported in the rules for **load** and **store** instructions. This would require to evaluate the symbolic expression in the context of a symbolic state and a path predicate ϕ, and then compute all concrete addresses as for the **cjmp** rules. This process can in general be infeasible due to the huge amount of possible concrete addresses at a

$$\frac{\Pi(pc) = \mathbf{cjmp}(e_1, e_2, e_3) \quad \Delta^s, \mu^s \vdash e_1 \Downarrow e_1^s \quad \phi' = (\phi \wedge e_1^s \neq 0) \quad \Delta^s, \mu^s \vdash e_2 \Downarrow pc^s \quad \phi' \wedge (pc^s = pc') \text{ consistent}}{(\Delta^s, \mu^s, \phi, pc, t) \rightarrow (\Delta^s, \mu^s, \phi', pc', t')}$$

$$\frac{\Pi(pc) = (reg := e) \quad \Delta^s, \mu^s \vdash e \Downarrow e^s}{(\Delta^s, \mu^s, \phi, pc, t) \rightarrow (\Delta^s[reg \mapsto e^s], \mu^s, pc + 1, t')}$$

$$\frac{\Pi(pc) = \mathbf{store}(e_1, e_2) \quad \Delta^s, \mu^s \vdash e_1 \Downarrow e_1^s \quad \Delta^s, \mu^s \vdash e_2 \Downarrow e_2^s}{(\Delta^s, \mu^s, \phi, pc, t) \rightarrow (\Delta, \mu^s[e_1^s \mapsto e_2^s], \phi, pc + 1, t')}$$

$$\frac{\Pi(pc) = \mathbf{assume}(e) \quad \Delta^s, \mu^s \vdash e \Downarrow e^s \quad \phi' = (\phi \wedge e^s \neq 0)}{(\Delta^s, \mu^s, \phi, pc, t) \rightarrow (\Delta^s, \mu^s, \phi', pc + 1, t')}$$

$$\frac{\Pi(pc) = \mathbf{cjmp}(e_1, e_2, e_3) \quad \Delta^s, \mu^s \vdash e_1 \Downarrow e_1^s \quad \phi' = (\phi \wedge e_1^s = 0) \quad \Delta^s, \mu^s \vdash e_3 \Downarrow pc^s \quad \phi' \wedge (pc^s = pc') \text{ consistent}}{(\Delta^s, \mu^s, \phi, pc.t) \rightarrow (\Delta^s, \mu^s, \phi', pc', t')}$$

$$\frac{\Pi(pc) = \mathbf{assert}(e) \quad \Delta^s, \mu^s \vdash e \Downarrow e^s \quad \models (\phi \Rightarrow e^s \neq 0)}{(\Delta^s, \mu^s, \phi, pc.t) \rightarrow (\Delta^s, \mu^s, \phi, pc + 1, t')}$$

$$\frac{\Pi(pc) = \mathbf{assert}(e) \quad \Delta^s, \mu^s \vdash e \Downarrow e^s \quad \not\models (\phi \Rightarrow e^s = 0)}{(\Delta^s, \mu^s, \phi, pc, t) \rightarrow \bot}$$

Figure 6.2: Symbolic Semantics of Instructions, where $t' = t + \tau(\Pi(pc))$

given point, most of which will be irrelevant to the final analysis. The solution we adopt in Fig. 6.2 is to propagate the symbolic expression and postpone the address resolution when needed. The **assert** rules use a first order oracle to decide the validity of the asserted expression, while the **assume** rule propagates the constraint as expected. Another problem that arises with the proof system is the possible nontermination due to unbounded loops, which we tackle by providing invariants, as discussed later.

The correctness of forward symbolic execution can be justified in terms of the strongest postcondition transformer [103]. We start with a SiML program Π and a property vector F, both having the same length. The property vector assigns to each program location l a formula F_l, which represents a property of executions reaching l. The strongest postcondition vector $sp(\Pi, F)$ consists of entries $sp(\Pi, F)_l$ representing the pointwise strongest, i.e. smallest, condition which guarantees that, when property F_j holds in the prestate and control passes from in-

struction j to l, then $sp(\Pi, F)_l$ holds in the poststate. The construction uses the iterator $spstep(\iota, \phi, j, l)$ which handles the case of control transfers from j, with ϕ holding at j, to l, with ι the instruction being executed. The sp function is pointwise monotone, and hence, using standard techniques, the largest cumulative fixed point F_{lim} satisfying $F_{lim} = sp(\Pi, F_{lim}) \sqsubseteq F_{init}$ can be obtained from an initial property vector F_{init}. The iterative computation is evidently not guaranteed to terminate, but, by choosing the F_{init} vector in an intelligent way and providing invariants, it is in fact possible to compute $sp(F_{init})$ in many concrete situations [93], even for programs with convoluted control flow.

6.5 Relational Symbolic Analysis

We now turn to the relational analysis for proving information flow properties defined in the previous sections. The main idea is to perform forward symbolic execution on a pair of programs and verify the information flow relation at each observation point.

6.5.1 Symbolic Observation Trees

The threat model assumes the attacker has access to part of the memory, can observe any store on his memory addresses and count the time elapsed up to the point where an observation occurs. The symbolic analysis accounts for the observation points, which are represented as symbolic constraints. We use a predicate P_O to define the range of the observable memory addresses. For instance, in Example 6.1, the untrusted agent has assigned memory addresses higher than 2K, i.e., $P_O(v) = v \geq 2048$, hence an explicit enumeration is quite expensive. Tracking dependencies on observable memory is tricky because the store instructions can be symbolic and thus potentially write to both observable and unobservable addresses. Therefore it is necessary to distinguish between observable stores, which affect the attackers state and unobservable stores, which do not affect the attackers state. We solve the issue by forking the symbolic execution engine each time we consider a store instruction. As reported in Fig. 6.3, we first evaluate address exp_1 and expression exp_2 in the symbolic context, and then distinguish between stores at observable addresses and stores at unobservable addresses. The predicate P_O partitions the symbolic execution into one branch where the store is *always* observable and one where the store is *always* unobservable. This process is important to guarantee the correctness of the entire approach.

The first rule captures the paths where the store instruction only affects observable addresses and thus is relevant for the subsequent security analysis. The second rule captures the paths where the store instruction affects the unobservable addresses, hence the analysis proceeds normally. The first rule is used to extract a *symbolic observation* tuple.

$$\frac{\Pi(pc) = \mathbf{store}(exp_1, exp_2) \quad \Delta^s, \mu^s \vdash exp_1 \Downarrow exp_1^s \quad \Delta^s, \mu^s \vdash exp_2 \Downarrow exp_2^s}{(\Delta^s, \mu^s, \phi, pc, t) \rightarrow (\Delta, \mu^s[exp_1^s \mapsto exp_2^s], (\phi \wedge P_O(exp_1^s)), pc + 1, t + \tau(\Pi(pc)))}$$

$$\frac{\Pi(pc) = \mathbf{store}(exp_1, exp_2) \quad \Delta^s, \mu^s \vdash exp_1 \Downarrow exp_1^s \quad \Delta^s, \mu^s \vdash exp_2 \Downarrow exp_2^s}{(\Delta^s, \mu^s, \phi, pc, t) \rightarrow (\Delta, \mu^s[exp_1^s \mapsto exp_2^s], (\phi \wedge \neg P_O(exp_1^s)), pc + 1, t + \tau(\Pi(pc)))}$$

Figure 6.3: Observable and Unobservable Stores

Definition 6.5.1 (Symbolic observation) *Consider a symbolic configuration* $C^s = (\Pi, \Delta^s, \mu^s, \phi, pc, t)$ *such that* $\Pi(pc) = \mathbf{store}(exp_1, exp_2)$. *Then a symbolic observation is the tuple* $obs = (\phi, exp_1^s, exp_2^s, t)$ *obtained after applying the first rule in Fig. 6.3.*

Intuitively, a symbolic observation captures how the concrete executions, starting from initial states that satisfy ϕ, affect the observable memory when they reach control point *pc*. This is done by recording the execution timestamp t and the possible values (exp_2^s) stored in each observable address (exp_1^s).

Alg. 2 tracks all symbolic observations occurring in the program by building a symbolic observation tree for a starting configuration C_0^s and a range predicate P_O. We use **fse** to represent all configurations produced in one step by the unary analysis described in Sect. 6.4. In particular, we assume that for store instructions, the first element of the output of **fse** corresponds to the first rule in Fig. 6.3 (which uses P_O). The procedure evaluates program instructions one by one and creates a tree node each time it reaches an observable store (line 10). The algorithm starts with the tree root *T.Start*, the observation range predicate P_O and the worklist containing the initial configuration C_0^s. For each statement, the algorithm updates the symbolic states by calling **fse** (line 7 and 17). We discard all symbolic states that have a non feasible path (line 5). When considering a store (line 6-13), the symbolic execution applies the rules in Fig. 6.3 and potentially produces an observable state C_o^s and an unobservable state C_u^s. If C_o^s is feasible, (line 9), the algorithm creates a new node containing the symbolic observation and attaches it to the current tree node. Subsequent observations will be attached to the freshly created node. If a **halt** statement is reached (line 14-15), the current branch is terminated, else the worklist is updated with all symbolic states obtained by executing the non-store instruction (line 16-19).

We prove correctness of Alg. 2 only for bounded programs and discuss generalizations in Sect. 6.5.4. In a nutshell, we show that the tree produced by the algorithm contains all the information needed to verify the security of the original program Π. The symbolic observation tree T contains the observation traces induced by the model of Π. Given a concrete initial configuration C_0, the trace is extracted by considering a path in T (the node sequence from *Start* to *End*) such that C_0 satisfies all path predicates and the observation trace is obtained by the

Algorithm 2 Program to Symbolic Observation Tree

INPUT: Predicate P_O, C_0^s
OUTPUT: Tree T
1. $W := [(C_0^s, \mathit{createNode}(\mathit{Start}))]$
2. **While** $W \neq \emptyset$
3. pop (C^s, T) from W
4. $(\Pi, \Delta^s, \mu^s, \phi, pc, t) := C^s$
5. **If** ϕ is satisfiable
6. **If** $\Pi(pc) == \mathbf{store}(e_1, e_2)$
7. $(C_o^s, C_u^s) := \mathbf{fse}(C^s, P_O)$
8. $(\Pi, \Delta_o^s, \mu_o^s, \phi_o, pc_o, t_o) := C_o^s$
9. **If** ϕ_o is satisfiable
10. $\mathit{TreeN} :=$ createNode$(\mathit{obs}(\phi_o, (\Delta^s, \mu^s \vdash e_1), (\Delta^s, \mu^s \vdash e_2), t_o))$
11. push TreeN to $T.\mathit{children}$
12. push (C_o^s, TreeN) to W
13. push (C_u^s, T) to W
14. **Else If** $\Pi(pc) == \mathbf{halt}$
15. push $\mathit{createNode}(\mathit{End})$ to $T.\mathit{children}$
16. **Else**
17. **for** $C_1^s \in \mathbf{fse}(C^s, P_O)$
18. push (C_1^s, T) to W

evaluation of symbolic observations in C_0, for each node along the path.

Theorem 6.5.1 *Consider a SiML program* Π *and the corresponding symbolic observation tree* T*. Then, the set of observation traces of the model* $\mathcal{M}_\Pi$ *of* Π *and the set of observation traces of* T *are the same.*

6.5.2 Relational Analysis

Relational analysis relates two symbolic observations by means of a relation Ψ which we call a *connector*. A connector Ψ is a predicate over free variables of the symbolic observation pair. In our setting, the connector Ψ forces the equality of initial observable parts of the memory. Moreover, the connector can cope with declassification policies with no additional effort [38]. Indeed, one can refine the initial state indistinguishability relation in Def. 6.2.1 and Ψ, accordingly, to express the declassification policy.

Definition 6.5.2 (Relational Validity) *Consider* $\mathit{obs}_1 = (\phi_1, e_{1,1}^s, e_{1,2}^s, t_1)$ *and* $\mathit{obs}_2 = (\phi_2, e_{2,1}^s, e_{2,2}^s, t_2)$*, two symbolic observations, and a connector* Ψ*. The triple*

(obs_1, obs_2, Ψ) *is* relationally valid *if*

$$\mathcal{R} := (\Psi \wedge \phi_1 \wedge \phi_2 \wedge P_O(e^s_{1,1}) \wedge P_O(e^s_{2,1})) \Rightarrow (e^s_{1,1} = e^s_{2,1} \wedge e^s_{1,2} = e^s_{2,2} \wedge t_1 = t_2) \text{ is valid}$$

Relational validity is a key property for enforcing the security condition over traces. It basically states that a pair of symbolic observations is secure if for any execution pair which initially agrees on observable memory (enforced by the connector Ψ) and reaches the observable program points (enforced by the path conditions ϕ_1, ϕ_2), if the stores are performed on observable memory (enforced by $P_O(e^s_{1,1}), P_O(e^s_{2,1})$), then they write the same values ($e^s_{1,2} = e^s_{2,2}$) at the same observable addresses ($e^s_{1,1} = e^s_{2,1}$) at the same time ($t_1 = t_2$). This implies that the observable memory is not affected by changes on the secret memory, hence the program is secure wrt. that observation pair.

Relational symbolic analysis on symbolic trees is described in Alg. 3. The algorithm takes as input a tree T, a copy T' with all variables renamed and a connector Ψ which defines the relation between variables, in the usual style of self-composition [51]. It then calls the procedure *ValidityCheck* which visits the tree per levels and checks relational validity for each observation pair (the Cartesian product on sets of nodes $T(l)$ and $T'(l)$ in line 1-2). It is worth noting that the *End* node is considered as a special observation, which corresponds to normal termination. A first order oracle is used to determine the validity of the condition $\mathcal{R}$. If $\mathcal{R}$ is not valid, the oracle returns a counterexample. This corresponds to a pair of concrete initial states giving rise to a pair of concrete executions that falsify the security condition, i.e. a security attack. Otherwise, if all pairs are valid for all levels, then the program is secure.

Theorem 6.5.2 (Tree Security) *Let T be a symbolic observation tree and $\mathcal{M}_T$ the set of observation traces associated with T. Then $\mathcal{M}_T$ is observational deterministic if Alg. 3 returns Valid in line 6.*

Theorem 6.5.3 (Security) *Let Π be an SiML program and T the symbolic observation tree obtained by running Alg. 2 on Π . Let also Alg. 3 return* Valid *on input T and connector Ψ. Then $\mathcal{M}_\Pi$ is observational deterministic if Alg. 3 returns Valid in line 6.*

Example 6.5.1 *We demonstrate our approach using the program in Fig. 6.1 and omit the equivalent SiML program. Suppose all memory addresses higher than 2KB are observable by the attacker. That is $P_O(v) = (v \geq 2048)$. Algorithm 2 yields the symbolic observation tree depicted in Fig. 6.4. Each root-leaf path represents observation traces of the program. The first branch is introduced by the line 7 of the algorithm: the left path is taken if the address updated by instruction 0x0fc is observable, otherwise the right path is taken. The second branch is introduced by the conditional instruction 0x108, which updates the register* `R2` *only if the content*

Algorithm 3 Relational Verification on SOTs

INPUT: Symbolic Tree pair *T, T'*, Connector Ψ
OUTPUT: Secure or Insecure + Attack
1. *level* := 1
2. **Call** ValidityCheck(*T(l), T'(l),* Ψ)

ValidityCheck(*T(l), T'(l),* Ψ)
1. **For all** (TreeN, TreeN') in $T(l) \times T'(l)$
2. $((\phi_1, e^s_{1,1}, e^s_{1,2}, t_1), (\phi_2, e^s_{2,1}, e^s_{2,2}, t_2)) :=$ *(TreeN.obs, TreeN'.obs)*
3. $A := Valid(\Psi \wedge \phi_1 \wedge \phi_2 \wedge P_O(e^s_{1,1}) \wedge P_O(e^s_{2,1})) \Rightarrow (e^s_{1,1} = e^s_{2,1} \wedge e^s_{1,2} = e^s_{2,2} \wedge t_1 = t_2))$
4. **If** Invalid(A) **return** *Insecure, A*
5. ValidityCheck(*T(l+1), T'(l+1),* Ψ)
6. **return** *Valid*

of the memory at the address 1024 is zero (since the instruction contains the EQ suffix).

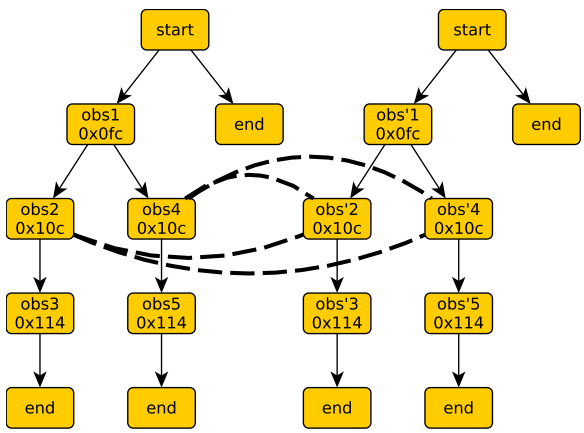


Figure 6.4: Symbolic Observation Trees

Suppose Alg. 2 has started with an initial symbolic configuration that bounds the i-th register to the fresh variable `Ri` *and the memory to the fresh variable M. The observations introduced by the instruction 0x10c are* obs_2 *and* obs_4 *according to the branch taken by the conditional instruction 0x108:*

$$obs_2 = ((M(R3) \geq 2048 \wedge M(1024) = 0), M(R3), 1)$$

$$obs_4 = ((M(R3) \geq 2048 \wedge M(1024) \neq 0), M(R3), 0)$$

Algorithm 3 takes the symbolic tree T and a copy T' with all variables in symbolic observations renamed, say obs'_i, and the connector Ψ. Assuming that the registers `R1` *and* `R3` *do not contain secret information, then $\Psi :=$* `R1` $= R1'$ $\wedge$ `R3` $=$ `R3'` $\wedge (\forall v.P_O(v) \Rightarrow M(v) = M'(v))$. *The symbolic tree has four levels (excluding the root) and the Cartesian product leads to 16 cases to be considered (i.e. four for each level).*

We consider the second level of the tree and the corresponding observations obs_2 and obs_4 and obs'_2 and obs'_4 as depicted in Fig. 6.4. In particular, $\mathcal{R}_{2,2} = (\Psi, obs_2, obs'_2)$ is relationally valid, while $\mathcal{R}_{2,4} = (\Psi, obs_2, obs'_4)$ is not. For instance, if `R3` $=$ `R3'` $= 2048$, $M(2048) = M'(2048) = 2052$, $M(1024) = 0$ *and $M'(1024) = 1$ then $\mathcal{R}_{2,4}$ is false. In fact, the program writes into the observable address* 2052 *different values depending on the content of the secret memory address* 1024.

6.5.3 Instantiation

Relational validity requires proving the validity of the following predicate $\mathcal{R} = (\Psi \wedge \phi_1 \wedge \phi_2 \wedge P_O(e^s_{1,1}) \wedge P_O(e^s_{2,1})) \Rightarrow$
$(e^s_{1,1} = e^s_{2,1} \wedge e^s_{1,2} = e^s_{2,2} \wedge t_1 = t_2)$ where the free variables can include the variables used to represent the initial registers and memories (we write R_i, R'_i, M and M' for registers and memories respectively). A connector Ψ is the predicate that forces the (relational) equality of initial observable parts of the memory and the equality of registers that do not contain secret information, that is $\Psi := R_1 = R'_1 \wedge \cdots \wedge (\forall v.P_O(v) \Rightarrow M(v) = M'(v))$. The resulting formula is clearly not quantifier free, hence it may result difficult for automatic theorem provers. This mainly depends on the observable predicate P_O which defines the range of observable addresses. If the range is small, one can simply enumerate the addresses and introduce the constraint $\Psi = \bigwedge_{v \in P_O(v)}(M(v) = M'(v))$. Since this range can be up to 2^{32} concrete addresses (i.e. 4GB), we extract from $\mathcal{R}$ all expressions that correspond to memory accesses, $M(e)$, and instantiate e for v in Ψ. This is recursively repeated for e and for all expressions in $\mathcal{R}$. Clearly, the number of such expressions can be huge, but still bounded by the number of memory accesses in the program code.

We illustrate the instantiation process with an example. Let the $\mathcal{R}$ predicate to contain the expression $M(M(R1) + M(R2))$, then the instantiation includes the constraints (i) $P_O(R1) \Rightarrow M(R1) = M'(R1)$, (ii) $P_O(R2) \Rightarrow M(R2) = M'(R2)$ and (iii) $P_O(M(R1) + M(R2)) \Rightarrow M(M(R1) + M(R2)) = M'(M(R1) + M(R2))$. Namely, for all expressions in $\mathcal{R}$ which represent memory accesses, we generate a constraint stating that if the address is observable, then the initial memory values are the same. The constraints we generate are sufficient to conclude about relational validity.

6.5.4 Invariants

The approach presented so far may not terminate due to the unbounded loops that might occur in the program. We handle this issue by decorating program loops with loop invariants [93]. Let Π_{Loop} be the program slice corresponding to the loop and let the loop be uniquely identified by a pair (pc_i, pc_e). We remove all back edges to cut the loop, namely the edge from pc_e to pc_i and apply the transformations from [46], which allow to cut the loop in a sound manner.

Proving invariants in this fashion is not sufficient for relational analysis. The main reason is that the approach only accounts for state invariants and fails to capture the number of observations that might be produced in each loop iteration. Therefore, a naive application of invariants in Alg. 3 would be unsound. Moreover, state invariants may require proving functional correctness of the loop. In fact, if a variable is updated in the loop body and later it contributes to an observation, the invariant must be sufficiently strong to identify the exact value of the variable. This may not be needed for proving security, therefore we use relational invariants which are in general simpler.

We propose a modification of Alg. 2 and Alg. 3, which is correct for programs with *natural* loops [46] (no jumps escaping the loop body). The main idea is to enforce relational invariants during the analysis of Alg. 3 and ensure that the loop pair is executed the same number of times. This requires to prove that not only the invariant but also the equivalence of the branch condition pair is preserved at each iteration. We first modify Alg. 2 to create a tree T_{Loop}, which consists of the symbolic observation tree of the loop body, the branch condition B annotating the root node and the symbolic configurations C^s annotating the leaf nodes. The tree T_{Loop} is uniquely identified and represents an approximated model of traces of Π_{Loop}. Similarly, non-loop trees are extended with the corresponding symbolic configurations annotating their leaf nodes. As a result, we obtain a tree which can be a *normal* tree, i.e. labeled with symbolic observations and symbolic states on leaf nodes or a *loop* tree, which in addition contains the branch condition labeling the root node. At this point, it is possible to generalize Alg. 3 to handle loops by means of relational invariants Ψ. To illustrate this, consider a program $P := P_1; (\mathit{while}\ B\ \mathit{do}\ P_2;) P_3$ and the corresponding trees $T := T_1; T_2; T_3$ obtained as described above. Let T, T' be the input tree pair to Alg. 3, Ψ_1 be the initial connector, and Ψ_2 the relational invariant of the loop tree T_2. As for traditional invariant verification, we first check that the (relational) invariant Ψ_2 holds before the loop entry, i.e. $\Psi_1 \wedge C_1^s \wedge C_1^{'s} \Rightarrow \Psi_2$, and it is preserved by the loop body, i.e. $\Psi_2 \wedge C_2^s \wedge C_2^{'s} \Rightarrow \Psi_2'$. This is done using the symbolic configurations C_i^s, $C_i^{'s}$ from the leaves of the observation trees T_i, T_i', where Ψ_i' denotes the connector after the execution of the pair (T_i, T_i'). In addition, we enforce that $\Psi_1 \wedge C_1^s \wedge C_1^{'s} \Rightarrow (B \Leftrightarrow B')$ and $\Psi_2 \wedge C_2^s \wedge C_2^{'s} \Rightarrow (B \Leftrightarrow B')$ to ensure that the loops are executed the same number of times. Finally, Alg. 3 can be applied to the symbolic observation trees (T_1, T_1'), (T_2, T_2') and (T_3, T_3'), using the connectors Ψ_1, $(\Psi_2 \wedge B \wedge B')$ and $(\Psi_2 \wedge \neg B \wedge \neg B')$, respectively. Two different cases can be encountered during

a run of Alg. 3. If a pair of normal nodes is reached, the relational validity is checked as before. If a pair of a loop node and a normal node is reached, they must be inconsistent. These conditions make our approach compositional with respect to observation traces. The process is repeated recursively for all pairs of nodes and, if successfully verified, it guarantees the security condition in Def. 6.2.1. The following example illustrates the relational verification of the UART driver routine of a separation kernel. For sake of clarity, here we reason at the C level and describe the case study more in detail later.

Example 6.5.2 *This code snippet transforms a 32 bit integer* `n` *into a hexadecimal number and, at each iteration, notifies the UART by updating three observable addresses (line 8-10).*

```
void printf_hex(uint32_t n) {
1.    int h,  i = 32 / 4 - 1;
2.    do {
3.        h = (n >> 28); n <<= 4;
4.        if(h < 10) h += '0';
5.        else h += 'A' - 10;
6.        usart_registers *usart0 = USART0_BASE;
7.        while(usart0->tcr != 0){  ;  }
8.        buffer_out[0] = h;
9.        usart0->tpr = (uint32_t)buffer_out;
10.       usart0->tcr = 1;
11.    }while(i--); }
```

The internal loop, which we discuss later, implements a polling routine on register `tcr`*, which is externally modified whenever the UART is ready to receive the next digit. Let* $\Psi_1 = (n = n')$ *be the initial connector relation and* $\Psi_2 = (n = n' \wedge i = i')$ *be the relational invariant of the external loop. The connector* Ψ_1 *holds of line* 1 *by the assumption that* `n` *is low. Moreover no observations occur, hence the first part is secure. The connector* Ψ_2 *holds of the external loop (lines 11-2) since the value of* `h` *written at the low address* `buffer_out[0]` *only depends on* `n`*, which is low, while the next two observations are fixed constants. The loop iterates the same number of times since the value of* `i` *is preserved by the relational invariant* Ψ_2*. Finally, the initial connector* $(n = n')$ *and the symbolic state pair at the loop entry* $(i = i' = 7)$ *trivially entail* Ψ_2*. Observe that if* `n` *were a secret location , then* $\Psi_2 =$ `true`*, and the verification would fail. Indeed, it is possible an execution pair goes through lines 4 and 5, and writes different values in* `buffer_out[0]`*.*

It is worth noting that the proposed verification approach ensures termination-sensitive noninterference, even for time-insensitive attacker models. We didn't find the security condition restrictive in our case studies. However, for loops without observations, one can relax the requirement on equal number of loop iterations and ensure termination-insensitive noninterference.

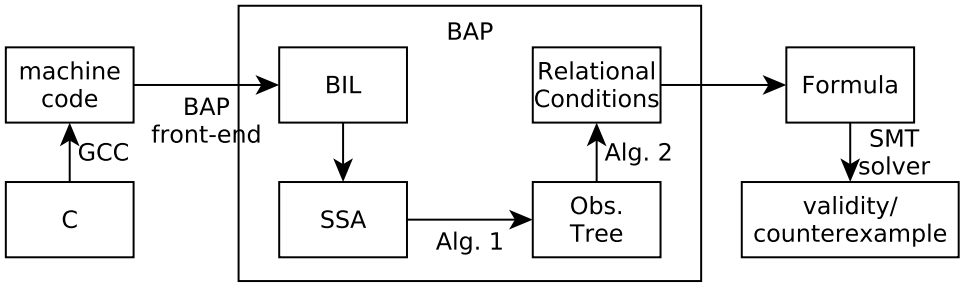


Figure 6.5: Verification process

6.6 Prototype Implementation

We implemented the relational analysis as a new back-end for the CMU Binary Analysis Platform framework [62]. The analogies between the BAP Intermediate Language (BIL) and SiML make the implementation of the prototype tool easier after enforcing minor syntactical constraints over the input programs. For instance, we do not allow multiple write accesses from a single instruction (i.e. each BIL instruction that writes in the memory always updates 4 bytes). Implementing the analysis as a new BAP back-end provided us several additional benefits: (i) the resulting prototype tool is architecture independent, (ii) there exists a verified transformer from ARMv7 assembly to BIL [93] and (iii) we can take benefit of the existing exporters to SMT solvers. We had to reimplement the symbolic execution engine of BAP v0.7 in order to handle our case studies. Variable substitution and conditional jumps constitute the main sources of exponential blowup and thus need special care. Fig. 6.5 depicts the workflow of the verification process. We start from GCC compiled machine code and lift it to the BIL language. The resulting BIL program is transformed into a single static assignment form (SSA) to enable efficient symbolic analysis and avoid the expensive substitution operation. In addition, several simplification and constant propagation routines have been implemented to further speed up the analysis. Loop invariants are currently provided manually and symbolic jumps are resolved statically. This phase produces the symbolic observation tree as described in Alg. 2. Subsequently, the relational analysis module generates quantifier-free formulas for a given pair of symbolic observation trees and a connector relation, as described in Alg. 3. Finally, using existing BAP exporters, the formulas are sent to the STP solver [118], which either validates the security property or provides a counterexample which violates the policy.

In the worst case, the size of observation tree can be exponential due to the well-known path explosion problem. We leverage standard optimizations such as expression substitution and constant propagation to reduce the tree size when possible. In addition, one can make use of the CFG to further control the exponential blow-up.

Software	ARM (LOC)	BIL (LOC)	Tool (Sec)	SMT (Sec)	Mem (Byte)	SOT (Nodes)	Sec (Y/N)
send syscall sender	80	1017	220	17	599M	31	Y
send syscall receiver	80	1017	220	16	599M	31	Y
send syscall sender 1	80	753	27	16	77M	31	N
send syscall sender 2	80	1015	215	18	599M	31	N
UART print char	13	100	1	1	85K	3	Y
UART print hex	32	305	5	1	29 M	7	Y
UART print bin	30	286	2	1	30M	7	Y
Exp. timing	19	151	1	1	319K	8	Y
Exp. timing	19	160	1	1	463K	7	N

Table 6.1: Experimental results

However, scalability issues are expected when fully verifying machine code. Consider symbolic jumps or symbolic memory. A bug-finding tool would simply concretize the symbols and continue the analysis. For verification one has to resolve all possible concretizations or carry the symbolic constraints throughout the analysis.

6.7 Case Studies

The tool has been used to verify several programs. In all cases, the analysis has been performed directly on the ARMv7-A machine code produced by the GCC compiler. The benchmark of these experiments is summarized in Table 6.1. Among other statistics we report the memory footprints and the number of SOT nodes to get an understanding of how big a piece of code we can currently check. Here, we summarize three case studies: (i) the IPC syscall of a separation kernel, (ii) a UART device driver and (iii) a modular exponentiation routine used by crypto services. The case studies are taken from real software. Our experience shows that small programs (order of 1000s instructions) are perfectly reasonable in high assurance contexts, and well within the reach of our tool with some standard engineering.

6.7.1 Case Study 1: Send syscall

The target separation kernel is a low level execution platform for ARMv7. The kernel implements minimal functionalities and consists of 1028 machine code instructions, mixing hand written assembly with GCC optimized output. The kernel must execute the partitions in isolation and control the communication appropri-

ately. Each partition is allowed to access a non-overlapping part of the system resources: (a) a contiguous part of the physical memory, that contains the partition's executable and data, (b) the logical message box, stored in the kernel memory and, (c) the virtual registers, which are stored in the kernel memory while the partition is suspended, and are stored in the standard registers while the partition is active.

The IPC mechanism is provided by the "send" syscall; first the active partition (the sender) stores the message in the register `R1` and raises a software interrupt, then the kernel handler stores the message in the message box of the receiver and restores the sender. While executing, the kernel backs up the sender's CPU state into its own memory and restores it when the syscall terminates. To appropriately control the communication, the kernel must ensure that: (1) the sender infers no information about the receiver and (2) the receiver only infers the content of sender's register R1 (the delivered message) and nothing more.

The above requirements have been verified by executing the relational analysis of the "send" syscall twice; considering observable the resources allocated to either the sender and the receiver. The resources allocated to the observing agent directly drive the definition of the connector relation (consisting in 30 lines of statements). Moreover, to take into account the designed declassification, the initial connector guarantees that the value of `R1` is equal in the two initial configurations. In the other experiments, we have modified the preconditions to test the tool with non-secure versions of the send syscall.

The absence of loop in the syscall freed us from defining the corresponding invariants. We also took benefit from the existing results obtained by a previous verification of functional correctness of the kernel: (i) the resolution of indirect jumps, (ii) the identification of data structures invariants and (iii) the analysis of constant parts of the memory.

These results reduced the set of reachable code of the syscall to 80 instructions (which corresponds to reducing the BIL code from 10K lines to 1017 lines) and provided us the necessary handler precondition (consisting of 400 lines of statements).

6.7.2 Case Study 2: UART device driver

The UART (Universal asynchronous receiver/transmitter) is a hardware device for communication over a serial interface. The driver is implemented in C and resembles the functionality of the well known `printf`. We limit our verification to the low level interface of the driver. Example 6.5.2 provides the C code that sends to the UART a 32 bit integer in the form of an hexadecimal number. The function contains two loops: the outer loop computes the eight hexadecimal digits of the number, the nested loop polls on the device register `tcr` before writing the current digit to the UART.

The function binary code consists of 32 instructions, that are lifted to 305 BIL lines. Initially the parameter `n` is represented by the register `R0` and the outer loop uses the register `R4` and `R5` to store the variable `n` and `i`, respectively. Since the

UART delivers the written characters to the external world, we consider observable the UART registers (64 bytes starting from `USART0_BASE`) and the DMA buffer (32 bytes starting from `buffer_out`). Moreover, since the input "must be sent" to the UART, we consider non secret the initial value of `R0`.

We first verified the nested loop. This fragment polls on the device register `tcr`, which is updated externally by the device driver. To emulate this external effect on the system memory, we inline the behavior of the device in the loop body. At each iteration a shadow variable `tcrWait` (which models an oracle knowing the number of iterations needed by the UART to receive a message) is decremented and `tcr` is resetted if the `tcrWait` value is zero. The given relational invariant `tcrWait=tcrWait'` states that the oracle provides the same answer in both executions. The tool automatically checks that the relational invariant is preserved and that the loop conditions are equivalent in both configurations. Notice that the nested loop does not produce observations, thus its verification is required only to guarantee termination sensitive noninterference.

Next we verify the outer loop. The relational invariant, `R4=R4' & R5=R5' & tcrWait=tcrWait'`, states that the values of `n`, `i` and the oracle answer are consistent in both configurations. The tool automatically checks that the relational invariant is preserved, the loop conditions are equivalent, the nested loop invariant is satisfied and the relational validity of the three observations (the update of the output buffer and the two UART registers). The relational verification is significantly simpler than the functional (total-)correctness of the loop; the relational invariant does not need to relate the values of `n` and `i`, the value of `h` is not constrained and no variant is needed.

The last verification step must ensure that starting from the initial connector Ψ, the condition of the outer loop is equivalent in both configurations and that the outer invariant is established. The initial connector Ψ simply relates the integer sent to the UART (the initial value of `R0`) and the content of the observable memory. The tool spotted that without further preconditions the code is not secure: before using the registers `R3`, `R4` and `R5` to represent the local variables, the function pushes their initial (secret) values on the stack. This yields an non (relationally) valid observation if the stack pointer is unconstrained.

6.7.3 Case Study 3: Modular exponentiation

The modular exponentiation routine is a simplified version of the case studies in [171]. The authors provide two programs with the same functionality. The insecure version branches on the secret boolean variable `i`, while the secure version computes the results independent of the branch condition, stores them in an array `A`, and returns `A[i]`. We point out that it is critical to verify this code at the machine level, as the compiler can perform optimizations that break the security.

We assume the attacker can only observe the execution time of the two routines. The elapse of time is modeled by a shadow variable, which is incremented for each instruction, following the functional time model described earlier. The tool detects

the control-flow side channel of the non-secure routine and validates the secure one, even if we do not require program counter equivalence between every pair of possible configuration.

6.8 Discussion and Related Work

Formal Verification of Low Level Code. Related kernel information flow verification efforts have been reported recently by several authors. For instance, [174] showed a noninterference property of the seL4 microkernel, essentially reducing to show absence of information flow from the scheduler to the next scheduled thread state. Similarly, [92] established system-level information flow security of a simple hypervisor at the level of ARMv7 assembly, by proving a trace equivalence property with respect to an ideal model that reflects the isolation properties that are desired of the hypervisor. Other machine code verification work includes the work by Heitmeyer et al. [133] and a series of works on the INTEGRITY kernel [190]. All these works use interactive theorem proving (ITP) techniques to establish the desired security property and consequently require serious manual effort. By contrast, this paper shows that automatic verification of small kernel routines is possible with less effort. Formal verification of device drivers has been applied to serial interfaces such as UART and USB devices. These works focus on functional correctness and ignore information flow properties. Moreover, the verification task is performed at C level [18, 172] or uses ITP [106].

Relational Verification. Relational program verification has been used to prove non-functional properties such as compiler optimization correctness [51], program equivalence [188, 183] and information flow security [52, 37, 149]. Neither addresses verification at the machine level. Barthe et al. [52] introduce self-composition as a method for checking 2-safety properties, including information flow. A related paper [51] presents product programs as a mean for reducing relational verification to classical functional verification. Several authors have studied algorithms for constructing and verifying over/under approximations of product programs automatically using typing [212], abstract interpretation [149, 183] and symbolic execution [37, 188, 206]. This work differs in several aspects. First, prior works do not consider timing channels and trace-based observations, giving rise to weaker security guarantees and simpler computational models. Second, we combine unary and relational analysis to avoid the expensive construction of the product program and reuse previously computed results. The unary analysis extracts necessary program dependences, while the relational one performs the verification. Breadth first search algorithms enable the alternation of these steps and allow efficient verification on the fly. As our case studies show, machine code verification requires flow and path sensitive techniques due to register reuse and complex data/control flow. Hence, compared to [149, 183], our techniques is more precise. Third, our approach addresses additional complications due to the lack of support for data structures. For instance, the secret state cannot be tied to program variables, and

it may depend on complex pointer arithmetic. Consequently, it is unknown a priori whether an instruction accesses a secret or public memory location. Recently, Caselden et al. [69] presented a way to recover a hybrid information flow/control flow graph using trace based analysis of machine code. This graph is used to find paths that trigger a given vulnerability condition. We find this work relevant and believe that their ideas can be applied to our setting and speed up the symbolic execution. However, the technique requires structural knowledge which one may not always have and ignores timing channels.

Timing. Eliminating timing channels by purely software approaches is difficult due to architecture dependent features such as caches, pipelines and more[15]. However, the timing constraints generated by our analysis can be used as a software contract to be enforced by hardware features. Our execution time model is history-independent and deterministic. We can not precisely represent the effect of caches and pipeline data dependencies. However, we can verify absence of side channels for simple architectures (e.g. ARM Cortex-M) or under the assumption that the attacker is not able to access to information that are affected by the wall-clock. Moreover, symbolic analysis provides memory access patterns which can be later validated wrt. a given architecture model. The model is similar to [131], which considers timing analysis for JavaCard-like bytecode. Molnar et al. [171] introduce the notion of PC-security which can avoid control flow side channels for crypto operations. Their security model can be easily accommodated in our work. Several authors propose mitigation [230] and padding [16] techniques to address timing channels. The results produced by our analysis are complementary to mitigation and padding. Indeed, they can be combined with mitigation to reduce the leakage bandwidth or to enable the required padding. We point out that worst case execution time is insufficient to verify that the execution time is independent of the secret, although it can remove information flows using mitigation. Finally, our model is suitable for systems where the external scheduler is instruction-based.

Loop Invariants. Finding loop invariants is definitely the most time consuming verification task and, for tricky examples, this is inherited by our works as well. However, relational invariants are in general simpler than functional invariants. We only need to enforce that the loop pair has the same low memory effects, without saying what these effects are. Recent work considers automatic generation of relational loop invariants for machine code using data driven techniques [206]. After executing the programs a certain number of times, concrete memory and register values are used to determine linear equality relationships between variables. Our approach can be used to check if the inferred invariants are sufficient to enforce equivalence for traces. The fact that we consider traces makes automatic invariant generation harder since traces do not compose in general. In [203], Saxena et al. introduce loop-extended symbolic execution which relates number of iterations of different program loops. This can be used to make our trace analysis more precise, although it was not needed in our case studies.

Security Analysis For Machine Code. Security analysis for machine code is a well studied research area [33]. The majority of works focus on bug finding

techniques for malware analysis, vulnerability checking, automatic exploit generation and more [180, 31, 72, 64]. Typically, they use typing, taint analysis or lightweight symbolic execution to ensure good path coverage and still maintain scalability. Other works take a more formal approach to machine code verification [189]. All these approaches fail to capture the information flows considered in this paper. Our focus is on full verification of small kernel handlers and device drivers. We admit that scalability remains an issue for larger programs and the techniques used by cited works can improve our tool.

Information Flow Analysis. Information flow has been pervasively applied to software security using static and dynamic verification techniques [197, 153]. If applicable, security type systems (TS) would be very efficient. Unfortunately, none of our case studies can be handled with TSs, at least not without significant modification. There are several case where a TS approach would fail: (i) Low observations are memory writes of shape $M[R_i] = exp$, hence dataflow analysis is needed to determine the values of R_i to know which address is updated. (ii)TSs don't support low memory writes under high branches. (iii)Our case studies use preconditions that guarantee certain invariants (describing the execution context). Data/control flow analysis is needed to determine the observations enabled in those contexts. (iv)Since traces do not compose in general, a global analysis through the symbolic trees is needed. (v)Unreachable or semantically secure code is also problematic for TSs. (vi)Declassification can be challenging and the timing analysis may require the TS to perform symbolic computation.

The verification approach for trace-based information flow analysis of ARMv7 machine code is novel. We leverage symbolic execution to reduce relational verification to automatic theorem proving of quantifier-free formulas. We are not aware of any tool that performs full verification of information flows for machine code.

Self-Modifying Code. Our analysis requires the code to be non self-modifying. For programs executed in user space this property is usually enforced at run-time by the underlying OS, for instance by configuring as non-writable the virtual memory containing the program code. On the other hand, privileged code can dynamically change its behavior by writing into its instruction memory, changing the coprocessor registers that control the MMU or updating the page tables. Considering these events as observable enables our analysis to verify that privileged code is non self-modifying. This can be done by checking that the corresponding symbolic observation tree is empty. It also enables the use of relational analysis for low level code that does not reconfigure the memory layout, but it accesses protected memory areas in privileged mode (e.g. the send syscall accesses the message boxes stored in the kernel memory) or performs privileged instructions (e.g. the syscall accesses the ARM banked registers to back up and restore the CPU context of the interrupted partition).

6.9 Conclusions

We presented a novel approach to relational verification for machine code. A distinguishing feature of our proposal is the ability to precisely verify information flow properties in the presence of features like a preemptive execution environment and memory mapped devices. We have implemented a tool and verified several real world case studies, including separation kernel routines and device drivers. This shows that information flow analysis for security critical routines is not only important, but also feasible.

There are several challenges we leave out as future work. The technique introduced in Alg. 1 can be combined with a security type system to automatically infer and refine types. This would improve the relational analysis by using Alg. 2 only when the typing fails. Timing is particularly critical to apply our approach to real processor architecture. We also are confident that, due to their simplicity, relational invariant generation can be automated. Other future plans include engineering the tool and improving on symbolic execution.

Acknowledgments

The authors thank the anonymous reviewers for valuable comments. This work is supported by framework grant “IT 2010” from the Swedish Foundation for Strategic Research.

Bibliography

[1] Flow caml. http://www.normalesup.org/~simonet/soft/flowcaml/. Accessed: 2014-08-12.

[2] Flowsafe: Information flow security for the browser. https://wiki.mozilla.org/FlowSafe. Accessed: 2014-08-12.

[3] Heartbleed ssl flaw's true cost will take time to tally. http://www.eweek.com/security/heartbleed-ssl-flaws-true-cost-will-take-time-to-tally.html. Accessed: 2014-08-06.

[4] Heartbleed website. http://heartbleed.com/. Accessed: 2014-08-06.

[5] Heartbleed wikipedia. http://en.wikipedia.org/wiki/Heartbleed. Accessed: 2014-08-06.

[6] Information flow control for the web. https://distrinet.cs.kuleuven.be/software/FlowFox/. Accessed: 2014-08-12.

[7] Jif: Java + information flow. http://www.cs.cornell.edu/jif/. Accessed: 2014-08-12.

[8] Joana (java object-sensitive analysis) - information flow control framework for java. http://pp.ipd.kit.edu/projects/joana/. Accessed: 2014-08-12.

[9] Jsflow. http://chalmerslbs.bitbucket.org/jsflow/. Accessed: 2014-08-12.

[10] The open web application security project. https://www.owasp.org/index.php/Main_Page. Accessed: 2014-08-07.

[11] Programming with paragon. http://paragon.nowplea.se/. Accessed: 2014-08-12.

[12] Xda-developers:android. http://forum.xda-developers.com/wiki/XDA-Developers:Android. Accessed: 2014-08-06.

[13] Simplified heartbleed explanation. http://commons.wikimedia.org/wiki/File:Simplified_Heartbleed_explanation.svg#mediaviewer/File:Simplified_Heartbleed_explanation.svg, August 2014. Inkscape. Licensed under Creative Commons Attribution-Share Alike 3.0 via Wikimedia Commons.

[14] M. Abadi, A. Banerjee, N. Heintze, and J. Riecke. A Core Calculus of Dependency. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '99, pages 147–160. ACM Press, 1999.

[15] Onur Aciiçmez and Çetin Kaya Koç. Microarchitectural attacks and countermeasures. In *Cryptographic Engineering*, pages 475–504. 2009.

[16] Johan Agat. Transforming out timing leaks. In *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '00, pages 40–53. ACM, 2000.

[17] Pieter Agten, Nick Nikiforakis, Raoul Strackx, Willem De Groef, and Frank Piessens. Recent developments in low-level software security. In *Lecture Notes in Computer Science*, pages 1–16. Springer, 2012.

[18] Eyad Alkassar, Mark A. Hillebrand, Steffen Knapp, Rostislav Rusev, and Sergey Tverdyshev. Formal device and programming model for a serial interface. In *VERIFY*, 2007.

[19] Rajeev Alur, Pavol Cerný, and Swarat Chaudhuri. Model checking on trees with path equivalences. In *Proceedings of the 13th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS' 07, pages 664–678. Springer-Verlag, 2007.

[20] Torben Amtoft, Sruthi Bandhakavi, and Anindya Banerjee. A logic for information flow in object-oriented programs. In *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '06, pages 91–102. ACM, 2006.

[21] G.R. Andrews and R. P. Reitman. An axiomatic approach to information flow in programs. *ACM Transactions on Programming Languages and Systems*, 2 (1):56–76, 1980.

[22] Ajit Appari and M Eric Johnson. Information security and privacy in healthcare: current state of research. *International Journal of Internet and Enterprise Management*, 6(4):279, 2010.

[23] ARMv7-A architecture reference manual. http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0406c. URL http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0406c.

[24] Aslan Askarov and Stephen Chong. Learning is change in knowledge: Knowledge-based security for dynamic policies. In *Proceedings of the 25th IEEE Computer Security Foundations Symposium*, CSF '12.

[25] Aslan Askarov, Sebastian Hunt, Andrei Sabelfeld, and David Sands. Termination-insensitive noninterference leaks more than just a bit. In *13th European Symposium on Research in Computer Security*, ESORICS 2008, pages 333–348. Springer, 2008.

[26] Aslan Askarov and Andrew Myers. A semantic framework for declassification and endorsement. In *Proceedings of the 19th European Conference on Programming Languages and Systems*, ESOP'10, pages 64–84. Springer-Verlag, 2010.

[27] Aslan Askarov and Andrew C. Myers. Attacker control and impact for confidentiality and integrity. *Logical Methods in Computer Science*, 7(3), 2011.

[28] Aslan Askarov and Andrei Sabelfeld. Gradual release: Unifying declassification, encryption and key release policies. In *IEEE Symposium on Security and Privacy*, SP '07.

[29] Aslan Askarov and Andrei Sabelfeld. Tight enforcement of information-release policies for dynamic languages. In *Proceedings of the 22nd IEEE Computer Security Foundations Symposium*, CSF '09.

[30] Thomas H. Austin and Cormac Flanagan. Permissive dynamic information flow analysis. In *Proceedings of the 5th ACM SIGPLAN Workshop on Programming Languages and Analysis for Security*, PLAS '10, pages 3:1–3:12. ACM, 2010.

[31] Thanassis Avgerinos, Sang Kil Cha, Brent Lim Tze Hao, and David Brumley. Aeg: Automatic exploit generation. In *In Proceeding of the Network and Distributed System Security Symposium*, NDSS '11, 2011.

[32] Arthur Azevedo de Amorim, Nathan Collins, André DeHon, Delphine Demange, Cătălin Hriţcu, David Pichardie, Benjamin C. Pierce, Randy Pollack, and Andrew Tolmach. A verified information-flow architecture. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, pages 165–178. ACM, 2014.

[33] Gogul Balakrishnan and Thomas Reps. Wysinwyx: What you see is not what you execute. *ACM Transactions on Programming Languages and Systems*, 32 (6):23:1–23:84, 2010.

[34] Thomas Ball, Rupak Majumdar, Todd D. Millstein, and Sriram K. Rajamani. Automatic predicate abstraction of c programs. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, PLDI '01, pages 203–213. ACM, 2001.

[35] Musard Balliu. A logic for information flow analysis of distributed programs. In *Proceedings of the 18th Nordic Conference*, NordSec '13.

[36] Musard Balliu, Mads Dam, and Roberto Guanciale. Automating Information Flow Analysis of Low Level Code. In *Proceedings of the 2012 ACM conference on Computer and Communications Security*, CCS '14, 2014. To appear.

[37] Musard Balliu, Mads Dam, and Gurvan Le Guernic. ENCoVer: Symbolic Exploration for Information Flow Security. In *Proceedings of the 25th IEEE Computer Security Foundations Symposium*, CSF '12, pages 30–44. IEEE, June 2012.

[38] Musard Balliu, Mads Dam, and Gurvan Le Guernic. Epistemic Temporal Logic for Information Flow Security. In *Proceedings of the 2009 Workshop on Programming Languages and Analysis for Security*, PLAS '09, pages 6:1–6:12. ACM, 2011.

[39] Musard Balliu and Gurvan Le Guernic. ENCoVer, June 2012. URL `http://www.nada.kth.se/~musard/encover`. Software release.

[40] Musard Balliu and Isabella Mastroeni. A Weakest Precondition Approach to Active Attacks Analysis. In *Proceedings of the 2009 Workshop on Programming Languages and Analysis for Security*, PLAS '09, pages 59–71. ACM, 2009.

[41] Musard Balliu and Isabella Mastroeni. A Weakest Precondition Approach to Robustness. *Transactions on Computational Science X*, 10:261–297, 2010.

[42] Anindya Banerjee, Roberto Giacobazzi, and Isabella Mastroeni. What you lose is what you leak: Information leakage in declassification policies. In *Proceedings of the 23th International Symposium on Mathematical Foundations of Programming Semantics*, volume 1514 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2007.

[43] Anindya Banerjee and David A. Naumann. Stack-based access control and secure information flow. *Journal of Functional Programming*, pages 131–177, 2005.

[44] Anindya Banerjee, David A. Naumann, and Stan Rosenberg. Expressive declassification policies and modular static enforcement. In *IEEE Symposium on Security and Privacy*, SP '08.

[45] Anindya Banerjee, David A. Naumann, and Stan Rosenberg. Expressive Declassification Policies and Modular Static Enforcement. In *IEEE Symposium on Security and Privacy*, SP '08, pages 339–353. IEEE, 2008.

[46] Mike Barnett and K. Rustan M. Leino. Weakest-precondition of unstructured programs. In *Proceedings of the 6th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE '05, pages 82–87. ACM, 2005.

[47] Clark Barrett, Aaron Stump, and Cesare Tinelli. SMT-LIB Logics (Version 2), 2011. URL `http://goedel.cs.uiowa.edu/smtlib/logics.html`.

[48] Clark Barrett, Aaron Stump, and Cesare Tinelli. SMT-LIB Theories (Version 2)", 2011. URL `http://goedel.cs.uiowa.edu/smtlib/theories.html`.

[49] Gilles Barthe, Gustavo Betarte, Juan Diego Campo, Carlos Luna, and David Pichardie. System-level non-interference for constant-time cryptography. 2015. To appear.

[50] Gilles Barthe, Salvador Cavadini, and Tamara Rezk. Tractable enforcement of declassification policies. In *Proceedings of the 21st IEEE Computer Security Foundations Symposium*, CSF '08.

[51] Gilles Barthe, Juan Manuel Crespo, and César Kunz. Relational verification using product programs. In *Proceedings of the 17th International Conference on Formal Methods*, FM '11, pages 200–214. Springer-Verlag, 2011.

[52] Gilles Barthe, Pedro R. D'Argenio, and Tamara Rezk. Secure information flow by self-composition. *Mathematical Structures in Computer Science*, 21 (6), 2011.

[53] Gilles Barthe, David Pichardie, and Tamara Rezk. A certified lightweight non-interference java bytecode verifier. In *Proceedings of the 16th European Conference on Programming*, ESOP'07, pages 125–140. Springer-Verlag, 2007.

[54] A. Baskar, R. Ramanujam, and S. P. Suresh. Knowledge-based modelling of voting protocols. In *Proceedings of the 11th conference on Theoretical aspects of rationality and knowledge*, TARK '07, pages 62–71. ACM, 2007.

[55] D. E. Bell and L. J. LaPadula. Secure computer systems: Mathematical foundations and model. Technical Report M74-244, MITRE Corp., Badford, MA, 1973.

[56] K. J. Biba. Integrity considerations for secure computer systems. Technical Report EDS-TR-76-372, USAF Electronic System Division, Badford, MA, 1977.

[57] Aaron Bohannon, Benjamin C. Pierce, Vilhelm Sjöberg, Stephanie Weirich, and Steve Zdancewic. Reactive noninterference. In *Proceedings of the 2009 ACM conference on Computer and Communications Security*, CCS '09, pages 79–90. ACM Press, 2009.

[58] Gérard Boudol and Ilaria Castellani. Noninterference for concurrent programs and thread systems. *Theoretical Computer Science*, 281(1):109–130, 2002.

[59] David FC Brewer and Michael J Nash. The chinese wall security policy. In *IEEE Symposium on Security and Privacy*, SP '89.

[60] Niklas Broberg and David Sands. Paralocks: Role-based information flow control and beyond. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '10, pages 431–444. ACM, 2010.

[61] Jeremy Brown and Thomas F Knight Jr. A minimal trusted computing base for dynamically ensuring secure information flow. 2001.

[62] David Brumley, Ivan Jager, Thanassis Avgerinos, and Edward J. Schwartz. Bap: A binary analysis platform. In *Computer Aided Verification*, CAV '11, 2011.

[63] David Brumley, Ivan Jager, Edward J. Schwartz, and Spencer Whitman. The bap handbook. October 2013. URL `http://bap.ece.cmu.edu/doc/bap.pdf`.

[64] David Brumley, Hao Wang, Somesh Jha, and Dawn Xiaodong Song. Creating vulnerability signatures using weakest preconditions. In *Proceedings of the 20th IEEE Computer Security Foundations Symposium*, CSF '07.

[65] Roberto Bruttomesso, Morgan Deters, and Alberto Griggio. Main Track Results of the Satisfiability Modulo Theories Competition (SMT-COMP), 2011. URL `http://www.smtexec.org/exec/?jobs=856`.

[66] Jeremy Bryans, Maciej Koutny, Laurent Mazaré, and Peter Y. A. Ryan. Opacity generalised to transition systems. In *International Workshop on Formal Aspects in Security and Trust*, FAST '05.

[67] Tevfik Bultan. BDD vs. Constraint-Based Model Checking: An Experimental Evaluation for Asynchronous Concurrent systems. In *Proceedings of the 6th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, TACAS '00, pages 441–455. Springer, 2000.

[68] Michael Burrows, Martín Abadi, and Roger M. Needham. A logic of authentication. *ACM Transactions on Computer Systems*, 8(1):18–36, 1990.

[69] Dan Caselden, Alex Bazhanyuk, Mathias Payer, Stephen McCamant, and Dawn Song. Hi-cfg: Construction by binary analysis and application to attack polymorphism. In *19th European Symposium on Research in Computer Security*, ESORICS 2013, pages 164–181. Springer-Verlag, 2013.

[70] Matteo Centenaro, Riccardo Focardi, Flaminia L. Luccio, and Graham Steel. Type-based analysis of pin processing apis. In *14th European Symposium on Research in Computer Security*, ESORICS 2008, pages 53–68. Springer-Verlag, 2009.

[71] Pavol Cerný and Rajeev Alur. Automated analysis of java methods for confidentiality. In *Computer Aided Verification*, CAV '09.

[72] Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, and David Brumley. Unleashing mayhem on binary code. In *IEEE Symposium on Security and Privacy*, SP '12.

[73] Rohit Chadha, Stéphanie Delaune, and Steve Kremer. Epistemic logic for the applied pi calculus. In *Formal Techniques for Distributed Systems*, volume 5522 of *Lecture Notes in Computer Science*, pages 182–197. Springer Berlin/Heidelberg, 2009.

[74] Stephen Chong and Andrew C. Myers. Decentralized robustness. In *Proceedings of the 19th IEEE Workshop on Computer Security Foundations*, CSFW '06, pages 242–256. IEEE.

[75] Stephen Chong and Andrew C. Myers. Security policies for downgrading. In *Proceedings of the 2004 ACM conference on Computer and Communications Security*, CCS '04, pages 198–209. ACM, 2004.

[76] Stephen Chong and Andrew C. Myers. Language-based information erasure. In *Proceedings of the 18th IEEE Workshop on Computer Security Foundations*, CSFW '05, pages 241–254. IEEE, 2005.

[77] Tom Chothia, Yusuke Kawamoto, Chris Novakovic, and David Parker. Probabilistic point-to-point information leakage. In *Proceedings of the 26th IEEE Computer Security Foundations Symposium*, CSF '13, pages 193–205. IEEE, 2013.

[78] Andrey Chudnov, George Kuan, and David Naumann. Information flow monitoring as abstract interpretation for relational logic. In *Proceedings of the 27th IEEE Computer Security Foundations Symposium*, CSF '14. IEEE, July 2014. To appear.

[79] Ravi Chugh, Jeffrey A. Meister, Ranjit Jhala, and Sorin Lerner. Staged information flow for javascript. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 50–62. ACM, 2009.

[80] David Clark and Sebastian Hunt. Non-interference for deterministic interactive programs. In *International Workshop on Formal Aspects in Security and Trust*, FAST '08.

[81] David Clark, Sebastian Hunt, and Pasquale Malacaria. Quantitative analysis of the leakage of confidential data. *Electronic Notes in Theoretical Computer Science*, 59:238–251, 2002.

[82] David D Clark and David R Wilson. A comparison of commercial and military computer security policies. In *IEEE Symposium on Security and Privacy*, SP '87.

[83] Jack Clark. Many android apps reveal user data. ZDNet website, September30 2010. `http://www.zdnet.com/news/many-android-apps-reveal-user-data/470361`.

[84] Michael R Clarkson, Bernd Finkbeiner, Masoud Koleini, Kristopher K Micinski, Markus N Rabe, and César Sánchez. Temporal logics for hyperproperties. In *Principles of Security and Trust*, POST '14.

[85] Michael R. Clarkson and Fred B. Schneider. Hyperproperties. *Journal of Computer Security*, 18(6):1157–1210, 2010.

[86] E. S. Cohen. Information transmission in computational systems. *ACM SIGOPS Operating System Review*, 11(5):133–139, 1977.

[87] E. S. Cohen. Information Transmission in Sequential Programs. *Journal of Foundations of Secure Computation*, pages 297–335, 1978.

[88] Mika Cohen and Mads Dam. A complete axiomatization of knowledge and cryptography. In *IEEE Symposium on Logic in Computer Science*, LICS '07.

[89] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '77, pages 238–252. ACM Press, 1977.

[90] Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In *Proceedings of the 6th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '79, pages 269–282. ACM, 1979.

[91] Mads Dam. Decidability and proof systems for language-based noninterference relations. In *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '06, pages 67–78. ACM Press, 2006.

[92] Mads Dam, Roberto Guanciale, Narges Khakpour, Hamed Nemati, and Oliver Schwarz. Formal verification of information flow security for a simple arm-based separation kernel. In *Proceedings of the 2013 ACM conference on Computer and Communications Security*, CCS '13, pages 223–234. ACM, 2013.

[93] Mads Dam, Roberto Guanciale, and Hamed Nemati. Machine code verification of a tiny arm hypervisor. In *Proceedings of the 3rd International Workshop on Trustworthy Embedded Devices*, TrustED '13, pages 3–12. ACM, 2013.

[94] Mads Dam, Gurvan Le Guernic, and Andreas Lundblad. Treedroid: A tree automaton based approach to enforcing data processing policies. In *Proceedings of the 2012 ACM conference on Computer and Communications Security*, CCS '12, pages 894–905. ACM, 2012.

[95] Al Danial. CLOC: Count lines of code. `http://cloc.sourceforge.net`, 2011. version 1.55.

[96] S. Danicic, M. Harman, R. Hierons, J. Howroyd, and M. Laurence. In *Proceedings of the 1st International Workshop on the Programming Language Interference and Dependence*, PLID 2004.

[97] Ádám Darvas, Reiner Hähnle, and David Sands. A theorem proving approach to analysis of secure information flow. In *Proceedings of the Second International Conference on Security in Pervasive Computing*, SPC '05, pages 193–209. Springer-Verlag, 2005.

[98] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, 2008.

[99] D. E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–242, 1976.

[100] D. E. Denning and P. Denning. Certification of programs for secure information flow. *Communications of the ACM*, 20(7):504–513, 1977.

[101] Dominique Devriese and Frank Piessens. Noninterference through secure multi-execution. In *IEEE Symposium on Security and Privacy*, SP '10, pages 109–124. IEEE, 2010.

[102] E. W. Dijkstra. *A Discipline of programming*. Prentice-Hall, 1976.

[103] Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of The ACM*, 18(8):453–457, 1975.

[104] Rayna Dimitrova, Bernd Finkbeiner, Máté Kovács, Markus N. Rabe, and Helmut Seidl. Model checking information flow in reactive systems. In *Proceedings of the 13th International Conference on Verification, Model Checking, and Abstract Interpretation*, VMCAI '12, pages 169–185. Springer-Verlag, 2012.

[105] Danny Dolev and Andrew C. Yao. On the security of public key protocols. Technical report, Stanford University, Stanford, CA, USA, 1981.

[106] Jianjun Duan and John Regehr. Correctness proofs for device drivers in embedded systems. In *Proceedings of the 5th International Conference on Systems Software Verification*, SSV'10, pages 5–5. USENIX Association, 2010.

[107] Jérémy Dubreil. Opacity and Abstractions. In *Proceedings of the First International Workshop on Abstractions for Petri Nets and Other Models of Concurrency*, APNOC '09.

[108] Petros Efstathopoulos, Maxwell Krohn, Steve VanDeBogart, Cliff Frey, David Ziegler, Eddie Kohler, David Mazières, Frans Kaashoek, and Robert Morris. Labels and event processes in the asbestos operating system. In *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles*, SOSP '05, pages 17–30. ACM, 2005.

[109] William Enck. A study of android application security. 2011.

[110] William Enck, Peter Gilbert, Byung-gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation*, OSDI'10, pages 1–6. USENIX Association, 2010.

[111] Ulfar Erlingsson, Yves Younan, and Frank Piessens. Low-level software security by example. In *Handbook of Information and Communication Security*. Springer, 2010.

[112] Ronald Fagin, Joseph Y. Halpern, Yoram Moses, and Moshe Y. Vardi. *Reasoning about knowledge*. MIT Press, Cambridge, Mass., 1995.

[113] Jeffrey Stewart Fenton. Memoryless subsystems. *The Computer Journal*, 17 (2):143–147, 1974.

[114] R. Focardi and R. Gorrieri. A classification of security properties for process algebras. *Journal of Computer security*, 3(1):5–33, 1995.

[115] Cédric Fournet, Gurvan Le Guernic, and Tamara Rezk. A Security-Preserving Compiler for Distributed Programs. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*, CCS '09, pages 432–441. ACM, 2009.

[116] Peter Gammie and Ron van der Meyden. MCK: Model Checking the Logic of Knowledge. In *Computer Aided Verification*, CAV '04.

[117] Priya Ganapati. Study shows some android apps leak user data without clear notifications. WIRED website, September30 2010. `http://www.wired.com/gadgetlab/2010/09/data-collection-android/`.

[118] Vijay Ganesh and David L. Dill. A decision procedure for bit-vectors and arrays. In *Computer Aided Verification*, CAV '07, pages 519–531. Springer, 2007.

[119] R. Giacobazzi and I. Mastroeni. Abstract non-interference: Parameterizing non-interference by abstract interpretation. In *Proceedings of the 31st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '04, pages 186–197. ACM, 2004.

[120] Pablo Giambiagi and Mads Dam. On the secure implementation of security protocols. *Science of Computer Programming*, 50(1-3):73–99, 2004.

[121] Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 213–223. ACM, 2005.

[122] J. A. Goguen and J. Meseguer. Security policies and security models. In *IEEE Symposium on Security and Privacy*, SP '82, pages 11–20. IEEE, 1982.

[123] Joseph A. Goguen and José Meseguer. Unwinding and inference control. In *IEEE Symposium on Security and Privacy*, SP '84, pages 75–86. IEEE Computer Society, 1984.

[124] Ian Goldberg, David Wagner, Randi Thomas, and Eric A. Brewer. A secure environment for untrusted helper applications confining the wily hacker. In *Proceedings of the 6th Conference on USENIX Security Symposium*, SSYM '96. USENIX Association, 1996.

[125] James W. Gray, III and Paul F. Syverson. A logical approach to multilevel security of probabilistic systems. *Distributed Computing*, 11(2):73–90, 1998.

[126] David Gries. *The Science of Programming*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1st edition, 1987.

[127] Joshua D. Guttman and Mark E. Nadel. What needs securing. In *Proceedings of the 1st IEEE Workshop on Computer Security Foundations*, CSFW '88.

[128] Joseph Y. Halpern and Kevin R. O'Neill. Secrecy in multiagent systems. *ACM Transactions on Information and System Security*, 12(1):5:1–5:47, 2008.

[129] HATS project (FP7-231620). *Deliverable D4.1: Report on Security*, 2012. Chapter 2.

[130] Daniel Hedin and Andrei Sabelfeld. Information-flow security for a core of javascript. In *Proceedings of the 25th IEEE Computer Security Foundations Symposium*, CSF '12, pages 3–18. IEEE, 2012.

[131] Daniel Hedin and David Sands. Timing aware information flow security for a javacard-like bytecode. *Electronic Notes Theoretical Computer Science*, 141 (1):163–182, 2005.

[132] Eric C.R. Hehner. *The Logic of Programming*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1984.

[133] Constance L. Heitmeyer, Myla Archer, Elizabeth I. Leonard, and John Mclean. Formal specification and verification of data separation in a separation kernel for an embedded system. In *Proceedings of the 2006 ACM conference on Computer and Communications Security*, CCS '06, pages 346–355. ACM, 2006.

[134] Boniface Hicks, Dave King, Patrick McDaniel, and Michael Hicks. Trusted declassification:: High-level policy for a security-typed language. In *Proceedings of the 2006 Workshop on Programming Languages and Analysis for Security*, PLAS '06, pages 65–74. ACM, 2006.

[135] HOL4. `http://hol.sourceforge.net/`. URL `http://hol.sourceforge.net/`.

[136] Michael Howard and David E. Leblanc. *Writing Secure Code*. Microsoft Press, Redmond, WA, USA, 2nd edition, 2002. ISBN 0735617228.

[137] Catalin Hritcu, Michael Greenberg, Ben Karel, Benjamin C. Pierce, and Greg Morrisett. All your ifcexception are belong to us. In *IEEE Symposium on Security and Privacy*, SP '13.

[138] Marieke Huisman, Pratik Worah, and Kim Sunesen. A temporal logic characterisation of observational determinism. In *Proceedings of the 19th IEEE Workshop on Computer Security Foundations*, CSFW '06. IEEE, 2006.

[139] Sebastian Hunt and David Sands. On flow-sensitive security types. In *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '06, pages 79–90. ACM, 2006.

[140] Dongseok Jang, Ranjit Jhala, Sorin Lerner, and Hovav Shacham. An empirical study of privacy-violating information flows in JavaScript Web applications. In *Proceedings of the 2010 ACM conference on Computer and Communications Security*, CCS '10, pages 270–83. ACM Press, 2010.

[141] Ranjit Jhala and Rupak Majumdar. Software model checking. *ACM Computing Surveys*, 41(4), 2009.

[142] Rajeev Joshi and K. Rustan M. Leino. A semantic approach to secure information flow. *Science of Computer Programming*, 37(1-3):113–138, 2000.

[143] Bill Joy and Ken Kennedy. Information technology research: Investing in our future. *President's Information Technology Advisory Committee*, February 1999.

[144] Vineeth Kashyap, Ben Wiedermann, and Ben Hardekopf. Timing-and termination-sensitive secure information flow: Exploring a new approach. In *IEEE Symposium on Security and Privacy*, SP '11.

[145] Sarfraz Khurshid, Corina Pasareanu, and Willem Visser. Generalized symbolic execution for model checking and testing. In *Tools and Algorithms for the Construction and Analysis of Systems*, TACAS '03, pages 553–568. Springer Berlin / Heidelberg, 2003.

[146] James C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[147] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. sel4: Formal verification of an os kernel. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, pages 207–220. ACM, 2009.

[148] P. Kocher. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. *Crypto '96*, pages 104–113, 1996.

[149] Máté Kovács, Helmut Seidl, and Bernd Finkbeiner. Relational abstract interpretation for the verification of 2-hypersafety properties. In *Proceedings of the 2013 ACM conference on Computer and Communications Security*, CCS '13, pages 211–222. ACM, 2013.

[150] Dexter Kozen. Language-based security. In *Proceedings of the 24th International Symposium on Mathematical Foundations of Computer Science*, MFCS '99, pages 284–298. Springer-Verlag, 1999.

[151] B. Lampson. A note on the confinement problem. In *Communications of the ACM*, pages 613–615, New York, 1973. ACM-Press.

[152] Peeter Laud. Semantics and program analysis of computationally secure information flow. *Programming Languages and Systems*, pages 77–91, 2001.

[153] Gurvan Le Guernic. *Confidentiality Enforcement Using Dynamic Information Flow Analyses.* PhD thesis, Kansas State University, 2007.

[154] K. Rustan M. Leino. Efficient weakest preconditions. *Information Processing Letters*, 93(6):281–288, 2005.

[155] Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.

[156] Peng Li and Steve Zdancewic. Downgrading policies and relaxed noninterference. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '05, pages 158–170. ACM, 2005.

[157] Alessio Lomuscio, Hongyang Qu, and Franco Raimondi. MCMAS: A Model Checker for the Verification of Multi-Agent Systems. In *Computer Aided Verification*, CAV '09, pages 682–688. Springer Berlin / Heidelberg, 2009.

[158] Ingo Lutkebohle. Same origin policy for javascript. URL `https://developer.mozilla.org/En/Same_origin_policy_for_JavaScript`.

[159] Heiko Mantel. The framework of selective interleaving functions and the modular assembly kit. In *Proceedings of the 2005 ACM Workshop on Formal Methods in Security Engineering*, FMSE '05, pages 53–62. ACM, 2005.

[160] Heiko Mantel and David Sands. Controlled declassification based on intransitive noninterference. In *Asian Symposium on Programming Languages and Systems*, APLAS '04.

[161] Radu Mardare and Corrado Priami. Decidable extensions of hennessy-milner logic. In *Formal Techniques for Networked and Distributed Systems*, volume 4229 of *FORTE '06*, pages 196–211. Springer Berlin / Heidelberg, 2006.

[162] I. Mastroeni and A. Banerjee. Modelling declassification policies using abstract domain completeness. Technical Report RR 61/2008, Department of Computer Science, University of Verona, 2008.

[163] Isabella Mastroeni. On the role of abstract non-interference in language-based security. In *Asian Symposium on Programming Languages and Systems*, APLAS '05, pages 418–433. Springer-Verlag, 2005.

[164] John Mclean. A general theory of composition for trace sets closed under selective interleaving functions. In *IEEE Symposium on Security and Privacy*, SP '94.

[165] John McLean. Security models and information flow. In *IEEE Symposium on Security and Privacy*, SP '90.

[166] John McLean. Proving noninterference and functional correctness using traces. *Journal of Computer Security*, 1(1):37–58, 1992.

[167] John McLean. A general theory of composition for a class of "possibilistic" properties. *IEEE Transactions on Software Engineering*, 22(1):53–67, 1996.

[168] Ricardo Medel, Adriana B. Compagnoni, and Eduardo Bonelli. A typed assembly language for non-interference. In *ICTCS*, pages 360–374, 2005.

[169] Dimiter Milushev and Dave Clarke. Incremental hyperproperty model checking via games. In *Proceedings of the 18th Nordic Conference*, NordSec '13.

[170] Vebjorn Moen, Andre N. Klingsheim, Kent Inge Fagerland Simonsen, and Kjell Jorgen Hole. Vulnerabilities in e-governments. *International Journal Electronic Security and Digital Forensics*, 1(1):89–100, 2007.

[171] David Molnar, Matt Piotrowski, David Schultz, and David Wagner. The program counter security model: Automatic detection and removal of control-flow side channel attacks. In *Proceedings of the 8th International Conference on Information Security and Cryptology*, ICISC'05, pages 156–168. Springer-Verlag, 2006.

[172] David Monniaux. Verification of device drivers and intelligent controllers: A case study. In *Proceedings of the 7th ACM/IEEE International Conference on Embedded Software*, EMSOFT '07, pages 30–36. ACM, 2007.

[173] Toby Murray, Daniel Matichuk, Matthew Brassil, Peter Gammie, Timothy Bourke, Sean Seefried, Corey Lewis, Xin Gao, and Gerwin Klein. seL4: From general purpose to a proof of information flow enforcement. In *IEEE Symposium on Security and Privacy*, SP '13, pages 415–429. IEEE, 2013.

[174] Toby Murray, Daniel Matichuk, Matthew Brassil, Peter Gammie, and Gerwin Klein. Noninterference for operating system kernels. In *Proceedings of the Second International Conference on Certified Programs and Proofs*, CPP '12, pages 126–142. Springer-Verlag, 2012.

[175] Andrew C. Myers and Barbara Liskov. Protecting privacy using the decentralized label model. *ACM Transactions on Software Engineering and Methodology*, 9:410–442, 2000.

[176] Andrew C. Myers, Andrei Sabelfeld, and Steve Zdancewic. Enforcing robust declassification. In *Proceedings of the 17th IEEE Workshop on Computer Security Foundations*, CSFW '04. IEEE, 2004.

[177] Aleksandar Nanevski, Anindya Banerjee, and Deepak Garg. Dependent type theory for verification of information flow and access control policies. *ACM Transactions on Programming Languages and Systems*, pages 6:1–6:41, 2013.

[178] David A Naumann. From coupling relations to mated invariants for checking information flow. In *11th European Symposium on Research in Computer Security*, ESORICS 2006, pages 279–296. Springer, 2006.

[179] Florian Nentwich, Nenad Jovanovic, Engin Kirda, Christopher Kruegel, and Giovanni Vigna.

[180] James Newsome and Dawn Xiaodong Song. Dynamic taint analysis for automatic detection, analysis, and signaturegeneration of exploits on commodity software. In *In Proceeding of the Network and Distributed System Security Symposium*, NDSS '05, 2005.

[181] Nick Nikiforakis, Luca Invernizzi, Alexandros Kapravelos, Steven Van Acker, Wouter Joosen, Christopher Kruegel, Frank Piessens, and Giovanni Vigna. You are what you include: Large-scale evaluation of remote javascript inclusions. In *Proceedings of the 2012 ACM conference on Computer and Communications Security*, CCS '12, pages 736–747. ACM, 2012.

[182] Kevin R. O'Neill, Michael R. Clarkson, and Stephen Chong. Information-flow security for interactive programs. In *Proceedings of the 19th IEEE Workshop on Computer Security Foundations*, CSFW '06, pages 190–201. IEEE, 2006.

[183] Nimrod Partush and Eran Yahav. Abstract semantic differencing for numerical programs. In *Proceedings of the 20th International Static Analysis Symposium*, SAS '13.

[184] Corina S. Pasareanu and Neha Rungta. Symbolic pathfinder: Symbolic execution of java bytecode. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, ASE '10, pages 179–180. ACM, 2010.

[185] Corina S. Pasareanu, Neha Rungta, and Willem Visser. Symbolic execution with mixed concrete-symbolic solving. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ISSTA '11, pages 34–44. ACM, 2011.

[186] Alessandra Di Pierro, Chris Hankin, and Herbert Wiklicky. Approximate non-interference. *Journal of Computer Security*, 12:37–81, 2004.

[187] Franco Raimondi and Alessio Lomuscio. Automatic verification of multi-agent systems by model checking via ordered binary decision diagrams. *Journal of Applied Logic*, 5(2):235–251, 2007.

[188] David A. Ramos and Dawson R. Engler. Practical, low-effort equivalence verification of real code. In *Computer Aided Verification*, CAV '11, pages 669–685. Springer, 2011.

[189] Thomas W. Reps, Junghee Lim, Aditya V. Thakur, Gogul Balakrishnan, and Akash Lal. There's plenty of room at the bottom: Analyzing and verifying machine code. In *Computer Aided Verification*, CAV '10, pages 41–56. Springer, 2010.

[190] Raymond J. Richards. Modeling and security analysis of a commercial real-time operating system kernel. In *Design and Verification of Microprocessor Systems for High-Assurance Applications*, pages 301–322. Springer US, 2010.

[191] Bruno P. S. Rocha, Sruthi Bandhakavi, Jerry den Hartog, William H. Winsborough, and Sandro Etalle. Towards static flow-based declassification for legacy and untrusted programs. In *IEEE Symposium on Security and Privacy*, SP '10.

[192] A William Roscoe. Csp and determinism in security modelling. In *IEEE Symposium on Security and Privacy*, SP '95.

[193] Alejandro Russo, John Hughes, David Naumann, and Andrei Sabelfeld. Closing internal timing channels by transformation. In *Proceedings of the 11th Asian Computing Science Conference on Advances in Computer Science: Secure Software and Related Issues*, ASIAN'06, pages 120–135. Springer, 2007.

[194] Alejandro Russo and Andrei Sabelfeld. Dynamic vs. static flow-sensitive security analysis. In *Proceedings of the 23rd IEEE Computer Security Foundations Symposium*, CSF '10, pages 186–199. IEEE, 2010.

[195] Alejandro Russo, Andrei Sabelfeld, and Li Keqin. Implicit flows in malicious and nonmalicious code. In *Proceedings of the 2009 Marktoberdorf Summer School*.

[196] Andrei Sabelfeld and Heiko Mantel. Securing communication in a concurrent language. In *Proceedings of the 9th International Static Analysis Symposium*, SAS '02.

[197] Andrei Sabelfeld and Andrew C. Myers. Language-based information-flow security. *IEEE IEEE Journal on Selected Areas in Communications*, 21(1): 5–19, 2003.

[198] Andrei Sabelfeld and Andrew C. Myers. A model for delimited information release. In *Proceedings of the International Symposium on Software Security*, ISSS '03, pages 174–191. Springer-Verlag, 2004.

[199] Andrei Sabelfeld and Alejandro Russo. From dynamic to static and back: Riding the roller coaster of information-flow control research. In *Proceedings of the 7th International Andrei Ershov Memorial Conference on Perspectives of Systems Informatics*, PSI '09, pages 352–365. Springer, 2010.

[200] Andrei Sabelfeld and David Sands. Probabilistic noninterference for multithreaded programs. In *Proceedings of the 13th IEEE Workshop on Computer Security Foundations*, CSFW '00, pages 200–214. IEEE, 2000.

[201] Andrei Sabelfeld and David Sands. A per model of secure information flow in sequential programs. *Higher-Order and Symbolic Computation*, 14(1):59–91, 2001.

[202] Andrei Sabelfeld and David Sands. Declassification: Dimensions and principles. *Journal of Computer Security*, 17(5):517–548, 2007.

[203] Prateek Saxena, Pongsin Poosankam, Stephen McCamant, and Dawn Song. Loop-extended symbolic execution on binary programs. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis*, ISSTA '09, pages 225–236. ACM, 2009.

[204] Fred B. Schneider, J. Gregory Morrisett, and Robert Harper. A language-based approach to security. In *Informatics - 10 Years Back. 10 Years Ahead*, pages 86–101. Springer-Verlag, 2001.

[205] Koushik Sen, Darko Marinov, and Gul Agha. Cute: a concolic unit testing engine for c. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-13, pages 263–272. ACM, 2005.

[206] Rahul Sharma, Eric Schkufza, Berkeley Churchill, and Alex Aiken. Data-driven equivalence checking. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '13, pages 391–406. ACM, 2013.

[207] Geoffrey Smith. Probabilistic noninterference through weak probabilistic bisimulation. In *Proceedings of the 16th IEEE Workshop on Computer Security Foundations*, CSFW '06, pages 3–13. IEEE, 2006.

[208] Dawn Song, David Brumley, Heng Yin, Juan Caballero, Ivan Jager, Min Gyung Kang, Zhenkai Liang, James Newsome, Pongsin Poosankam, and Prateek Saxena. BitBlaze: A new approach to computer security via binary analysis. In *Proceedings of the 4th International Conference on Information Systems Security. Keynote invited paper*, Hyderabad, India, 2008.

[209] William Stallings and Lawrie Brown. *Computer Security: Principles and Practice*. Prentice Hall Press, Upper Saddle River, NJ, USA, 1st edition, 2007.

[210] G Edward Suh, Jae W Lee, David Zhang, and Srinivas Devadas. Secure program execution via dynamic information flow tracking. In *ACM SIGOPS Operating Systems Review*, volume 38, pages 85–96. ACM, 2004.

[211] David Sutherland. A model of information. In *9th National Computer Security Conference*, 1986.

[212] Tachio Terauchi and Alexander Aiken. Secure information flow as a safety problem. In *Proceedings of the 12th International Static Analysis Symposium*, SAS '05.

[213] Aditya V. Thakur, Junghee Lim, Akash Lal, Amanda Burton, Evan Driscoll, Matt Elder, Tycho Andersen, and Thomas W. Reps. Directed proof generation for machine code. In *Computer Aided Verification*, CAV '10, pages 288–305. Springer, 2010.

[214] Mohit Tiwari, Hassan MG Wassel, Bita Mazloom, Shashidhar Mysore, Frederic T Chong, and Timothy Sherwood. Complete information flow tracking from the gates up. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIV, pages 109–120. ACM, 2009.

[215] Ron van der Meyden and Chenyi Zhang. Algorithmic verification of noninterference properties. *Electronic Notes in Theoretical Computer Science*, 168: 61–75, 2007.

[216] Jeffrey A Vaughan and Steve Zdancewic. A cryptographic decentralized label model. In *IEEE Symposium on Security and Privacy*, SP '07, pages 192–206. IEEE, 2007.

[217] Willem Visser, Klaus Havelund, Guillaume Brat, SeungJoon Park, and Flavio Lerda. Model checking programs. *Automated Software Engineering*, 10:203–232, 2003.

[218] Dennis Volpano, Cynthia Irvine, and Geoffrey Smith. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(2-3):167–187, 1996.

[219] Dennis M. Volpano. Safety versus secrecy. In *Proceedings of the 6th International Symposium on Static Analysis*, SAS '99, pages 303–311. Springer-Verlag, 1999.

[220] Daniel Wasserrab, Denis Lohner, and Gregor Snelting. On PDG-Based Noninterference and its Modular Proof. In *Proceedings of the 2009 Workshop on Programming Languages and Analysis for Security*, PLAS '09, pages 31–44. ACM, 2009.

[221] David A. Wheeler. How to prevent the next heartbleed. `http://www.dwheeler.com/essays/heartbleed.html`. Accessed: 2014-08-06.

[222] Jerold Whitmore, Andre Bensoussan, Paul Green, Douglas Hunt, and Andrew Kobziar. Design for multics security enhancements. Technical report, DTIC Document, 1973.

[223] Glynn Winskel. *The Formal Semantics of Programming Languages: An Introduction*. MIT press, Cambridge, Mass., 1993.

[224] J. Todd Wittbold and Dale M. Johnson. Information flow in nondeterministic systems. In *IEEE Symposium on Security and Privacy*, SP '90, 1990.

[225] Andrew C. Yao. Protocols for secure computations. In *Proceedings of the 23rd Annual Symposium on Foundations of Computer Science*, SFCS '82, pages 160–164. IEEE, 1982.

[226] Steve Zdancewic. Challenges for information-flow security. In *Proceedings of the 1st International Workshop on the Programming Language Interference and Dependence*, PLID 2004.

[227] Steve Zdancewic and Andrew C. Myers. Robust declassification. In *Proceedings of the 14th IEEE Workshop on Computer Security Foundations*, CSFW '01, pages 5–. IEEE, 2001.

[228] Steve Zdancewic, Lantian Zheng, Nathaniel Nystrom, and Andrew C. Myers. Untrusted hosts and confidentiality: Secure program partitioning. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*, SOSP '01, pages 1–14. ACM, 2001.

[229] Nickolai Zeldovich, Silas Boyd-Wickizer, Eddie Kohler, and David Mazières. Making information flow explicit in histar. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7*, OSDI '06, pages 19–19. USENIX Association, 2006.

[230] Danfeng Zhang, Aslan Askarov, and Andrew C. Myers. Language-based control and mitigation of timing channels. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 99–110. ACM, 2012.

[231] Yajin Zhou and Xuxian Jiang. Dissecting android malware: Characterization and evolution. In *IEEE Symposium on Security and Privacy*, SP '12, pages 95–109. IEEE, 2012.

[232] Zongwei Zhou, Miao Yu, and Virgil D. Gligor. Dancing with giants: Wimpy kernels for on-demand isolated i/o. In *IEEE Symposium on Security and Privacy*, SP '14, pages 308–323. IEEE, 2014.