JSLINQ: Building Secure Applications across Tiers (Extended Version)

Musard Balliu
Chalmers

Benjamin Liebe
Chalmers

Daniel Schoepe
Chalmers

Andrei Sabelfeld
Chalmers

ABSTRACT

Modern web and mobile applications are complex entities amalgamating different languages, components, and platforms. The rich features span the application tiers and components, some from third parties, and require substantial efforts to ensure that the insecurity of a single component does not render the entire system insecure. As of today, the majority of the known approaches fall short of ensuring security across tiers.

This paper[1] proposes a framework for end-to-end security, by tracking information flow through the client, server, and underlying database. The framework utilizes homogeneous meta-programming to provide a uniform language for programming different components. We leverage .NET meta-programming capabilities from the F# language, thus enabling language-integrated queries on databases and interoperable heterogeneous execution on the client and the server. We develop a core of our security enforcement in the form of a security type system for a functional language with mutable store and prove it sound. Based on the core, we develop JSLINQ, an extension of the WebSharper library to track information flow. We demonstrate the capabilities of JSLINQ on the case studies of a password meter, two location-based services, a movie rental database, an online Battleship game, and a friend finder app. Our experiments indicate that JSLINQ is practical for implementing high-assurance web and mobile applications.

1. INTRODUCTION

There is no such thing as a free lunch - building secure and robust web applications is a complex and error prone task. A recurrent fact attested by investigations from security organizations and communities of security experts [8, 4], and very frequently reported by the media [9, 7], is that vulnerabilities in web and mobile applications dominate the classifications of the most dangerous security attacks. The reason can be attributed to different factors, including the myriad of programming languages, technologies and platforms which are used to build modern applications. This process requires substantial efforts and skills on the programmer's side for getting the application logic right, let alone secure and reliable. In this paper, we set out to study the challenge of heterogeneity and provide practical solutions with formal evidence, that help a programmer to build web and mobile applications in a secure manner. In particular, we focus on vulnerabilities that go beyond injection attacks and affect the business logic of the entire web application.

A closer look at a typical web architecture shows that web applications are often distributed over several tiers: (a) a client tier, where most of the UI logic runs in a web browser as JavaScript and HTML including third-party libraries; (b) a server tier, where the bulk of the application logic is executed in a language like F#, Java or other; (c) and a database tier that serves as persistent store and executes e.g. SQL code. Common security attacks rely on the fact that applications are implemented in different languages that span tiers with different trust relationships. As a result, many security policies are application-specific and tightly connected to the application logic and the trust relationships between the involved parties.

Motivating Scenarios: The following scenarios illustrate the need for cross-tier security analysis and policies.
Password Meter: The first scenario considers a client-side password meter, which is a program used to estimate the strength of passwords provided by users. It is important that the chosen password is not leaked to an application server or other third parties. A reasonable security policy treats the password field as sensitive, and the third-party and the RPC functions used to communicate with the application as public, while enforcing that no sensitive information flows to the public destinations.
Location-based Service: The second scenario is a location-based service, which uses location information to query a web service for the list of nearby points of interest, and a third-party map library to display these points. However, users concerned about privacy may not want to reveal the exact coordinates of their location. A reasonable security policy allows for a declassification function to obfuscate the real location, and only send approximate coordinates to the location server. Moreover, the map library should only be used to display the points of interest and not to, for instance, leak the browser's cookie to the library provider.
Friend Finder App: The third scenario is a mobile app. The user wants to know if a friend is using a certain app, say WhatsApp, without revealing the friend's phone number to the remote server in case they are not using that app. This can be avoided by using a hash function to hide the phone number before sending it to the database server, which in turn compares the hashed value to the list of its users' phone numbers and replies whether or not that user is using the app. A reasonable policy considers the phone address book as sensitive and ensures that only hashed values are sent to the untrusted application server for discovery.

These are all examples of how a security attack can occur across all three tiers of an application. Hence, a satisfactory security analysis needs to express and validate policies for applications that span client, server and database tiers.

Attacker Model: Different attacker models arise in multi-tier web applications. Sensitive or untrusted data may originate from any of the components, for instance it can be a

[1] This document is an extended version of the original paper with the same title, which got accepted at CODASPY'16.

user location from the client, a password from the database or an authentication key from the server. Consequently, any tier can be subject to unintentional or malicious information leaks toward another tier. The policies for the first two scenarios constrain the sensitive data of a trusted client wrt. an untrusted third-party library and a (partially) trusted server. The third scenario illustrates policies for a trusted client wrt. to a completely untrusted server. The client can also be untrusted. For example a trusted server, after authenticating a user, may read his personal data from a trusted database and send back a customized web page, however, no information about other users in the database should flow to the client. Meaningful combinations of tiers and attacker models will be discussed in Section 4. We do not address network attackers who intercept, alter or deny communication between tiers, while techniques like SSL can be used to prevent these types of attacks.

State of the Art: Information-flow control (IFC) tracks sensitive (untrusted) data throughout the computation ensuring that no illegal information flows from sensitive (untrusted) sources toward public (trusted) sinks. This provides end-to-end security guarantees as required in the scenarios above. In general, we mark sources and sinks with labels from a lattice of security levels that expresses the trust relationships between parties. E.g., horizontal privilege-escalation attacks can be prevented by assigning separate security labels for separate users. A large body of work has studied dynamic and static enforcement techniques for all levels of the hardware and software stack [22, 34], including web applications [26] and distributed systems [42]. The majority of these works tackles the problem of information flow for different components in isolation [38, 29, 23]. This is unsatisfactory because tracking information across tiers is necessary for end-to-end security. A few works, as discussed in Section 5, bridge IFC across components allowing for policies that regulate information flows for a web application as a whole. Noteworthy, recent frameworks integrate database queries into programming languages for client and server applications providing a uniform way to program an entire web application, including reasoning about security [18, 16, 15].

Contributions: In this paper we leverage homogeneous meta-programming to obtain a uniform language for reasoning about web and mobile application security across the client-server-database boundaries. The .NET facilities provide support for language-integrated queries on databases and interoperable heterogeneous execution for client and server applications, embedding them seamlessly in the F# language [40]. This allows to implement an entire web or mobile application as a simple F# program and then let the compiler split the code transparently for each tier. In this work we enrich a subset of the language with security types which allow to express security policies. We implement the security types by custom attributes as a separate F# module on top of existing fully-fledged development in F#, providing a complete separation between the program code and the security policy. We then execute the security type check as a separate verification step followed by the F# compilation and thus leaving the F# type system untouched. Finally, we split the program into three parts, producing JavaScript and HTML code to run on the browser, SQL code to run on the database and F# code to run on the server.

On the formal side (Section 2), we develop a model for a functional language with references (a subset of F#), quotations and antiquotations, and establish the soundness of the security type system. Our soundness proof extends and generalizes the proof technique introduced by Pottier and Simonet [30] with support for arbitrary data types and declassification policies. The query language is based on the one introduced by Cheney et al. [14] and uses quotation and normalization of quoted terms to model the semantics of the database language. For simplicity, our results assume a two-point security lattice for confidentiality, however, they apply to arbitrary lattices, including integrity, in a similar fashion.

On the practical side (Section 3), we have implemented JSLINQ, an extension of WebSharper [10] and LINQ [1] libraries with IFC. With JSLINQ, a developer can use a fully-fledged language such as F# for writing secure web and mobile applications. A security analyst is expected to know what sources and sinks are sensitive, which is a reasonable assumption so long as they are partially trusted. If the developer is malicious, one can leverage techniques from [27, 31] to automatically extract sources and sinks used by the application (this is out of scope in this work). The policy module requires to specify security signatures once and only for the APIs that are actually used, thus making it easier and less time-consuming for the programmer. Our experience shows that JSLINQ provides a good trade-off between annotation burden and security assurance for developers with some security background, while user studies with non-expert developers are subject to future work.

We demonstrate the capabilities of JSLINQ on several realistic case studies (Section 4), including the scenarios discussed above, a password meter and an online Battleship game. The case studies leave out user interfaces and other boilerplate code, and only focus on the security-critical parts of the applications to demonstrate the potential of our technique. Moreover, compositionality of the security type checking makes the approach scalable to arbitrary lines of code. The experiments show that JSLINQ is useful for building secure applications and it enjoys several advantages compared to existing tools (Section 5 and Table 2).

A precursor of our approach is SELINQ by Schoepe et al. [36]. SELINQ uses a security type system to enforce policies for server-database applications written in F#, as we do. Rather than enriching F# with security types, SELINQ implements a subset of the language presented in Section 2 and uses a compiler implemented in Haskell to type check and generate F# executable code. By contrast, JSLINQ closes the end-to-end loop by supporting client-side, including third-party code, for fully-fledged F# applications. A distinguishing feature of JSLINQ is that security type checking does not interfere with the normal development process. In practical terms, this translates to a big gain as the programmers can use a production-grade system to develop applications, yet leverage a security type system to verify the critical parts of the code. Moreover, practicality of JSLINQ is supported by several case studies and security policies. Declassification allows us to handle richer policies, e.g. only friends can view a user's profile data, while dynamic policies would require extending the type system with techniques from [43]. While both SELINQ and JSLINQ use the framework by Cheney et al. [14], JSLINQ significantly extends that formalism with mutable references and declassification using a different technique to show noninterference.

While our main focus is on multi-tier application-level attacks, JSLINQ inherits protection against XSS and SQL injection attacks from its components, respectively, from WebSharper and LINQ. Such attacks are impossible due to strong typing [32], similar to frameworks as GWT. For instance, an SQL injections are prevented by the use of LINQ, which leverages the underlying F# type system to strongly type all database queries.

The full details of the framework, including semantics and proofs, and the code for JSLINQ are available online [11].

2. FRAMEWORK

In this section we present the formal underpinnings of the framework. The client and the server components are written in the *host* language, while the database component is written in the *quoted* language. The framework consists of a functional language with mutable storage and support for product types, records, lists, quotations and antiquotations, the security type system, and shows that the type system enforces noninterference and declassification policies with respect to the operational semantics. The host and the quoted language represent a core of the F# language as implemented by JSLINQ.

2.1 Language

The language is presented in Figure 1. It includes the usual constructs of a functional language with references, extended with quotations and antiquotations to account for database queries. The syntax consists of security levels, types, and terms. $\overline{x}$ denotes a sequence of entities x.

$\ell ::= \ \mathtt{L} \mid \mathtt{H}$ (security types)

$b ::= \ \mathbf{bool}^{\ell} \mid \mathbf{int}^{\ell} \mid \mathbf{float}^{\ell} \mid \mathbf{string}^{\ell}$ (base types)

$t ::= \ b \mid \mathbf{unit} \mid t \overset{\ell}{\to} t \mid t \ \mathbf{ref}^{\ell} \mid t * t \mid \{\overline{f : t}\} \mid (t \ \mathbf{list})^{\ell} \mid \mathbf{Expr}\langle t \rangle$
(general types)

$T ::= \ (\{\overline{f : b}\}) \ \mathbf{list}^{\ell}$ (database tables)

$\Gamma, \Delta, M ::= \ \cdot \mid \Gamma, x : t \mid \Delta, x : t \mid M, l : t$ (type environment)

$e ::= \ () \mid c \mid x \mid l \mid op(\overline{e}) \mid \mathbf{lift}\ e \mid \mathbf{fun}(x) \to e$ (terms)
$\mid \ \mathbf{rec}\ f(x) \to e \mid (e, e) \mid \mathbf{fst}\ e \mid \mathbf{snd}\ e \mid \{\overline{f = e}\} \mid e.f$
$\mid \ \mathbf{yield}\ e \mid [] \mid e \ @\ e \mid \mathbf{for}\ x\ \mathbf{in}\ e\ \mathbf{do}\ e \mid \mathbf{exists}\ e$
$\mid \ \mathbf{if}\ e\ \mathbf{then}\ e\ \mathbf{else}\ e \mid \mathbf{if}\ e\ \mathbf{then}\ e \mid \mathbf{run}\ e \mid \mathtt{<@}\ e\ \mathtt{@>} \mid \mathtt{(\%}\ e\ \mathtt{)}$
$\mid \ \mathbf{database}(x) \mid \mathbf{ref}\ e \mid !e \mid e := e$

Figure 1: Syntax of language and types

We remark on some of the interesting constructs: c denotes built-in constants, such as booleans, integers, floats and strings. op denotes built-in operators, such as addition and logical connectives. **if** e_1 **then** e_2 **else** e_3 evaluates to e_2 if e_1 evaluates to **true** and to e_3 otherwise. The language includes mutable state. Terms **ref** e (reference creation), $!e$ (dereference) and $e := e$ (assignment) denote, respectively, allocating, dereferencing and updating memory locations. () denotes a value of type **unit**. Database queries are modelled by quoted expressions `<@` e `@>` of type $\mathbf{Expr}\langle t \rangle$. The language allows only closed quoted terms, since this simplifies the semantics of the language and is still able to express all the desired concepts. Quoted functions can be expressed by abstracting in the quoted term as opposed to abstracting on the level of the host language. `(%` e `)` denotes antiquotation of the expression e, and allows splicing of quoted expressions into quoted expressions in a type-safe way. **lift** e lifts an expression of type t to type $\mathbf{Expr}\langle t \rangle$. **for** x **in** e_1 **do** e_2 is used to express list comprehensions where x is bound successively to elements in e_1 when evaluating e_2. The results of evaluating e_2 for each element are then concatenated. **run** e denotes running a quoted expression e, which involves generating an SQL query based on the quoted term. e_1 @ e_2 denotes concatenation of e_1 and e_2. **exists** e evaluates to **true** if and only if the expression e does not evaluate to the empty list. This can be used to check if the result of a query is empty. **if** e_1 **then** e_2 evaluates to e_2 if e_1 evaluates to a non-empty list and to [] otherwise. **yield** e denotes a singleton list consisting of expression e.

Security type language: Security types are defined by annotating a standard type language for a functional fragment with quotations and references with security levels ℓ. The security levels are taken from the two-element lattice $\langle \{\mathtt{L}, \mathtt{H}\}, \sqsubseteq \rangle$ consisting of a level L for low-confidentiality (dually high-integrity) information and a level H for high-confidentiality (dually low-integrity) information. The ordering relation requires that $\mathtt{L} \sqsubseteq \mathtt{H}$. The types are split into base types (b), which can occur as types of columns in tables (T), and general types (t) which include unit, functions, references, tuples, records, lists, and quoted expression types. Function types include a level ℓ, which is a lower bound on the level of locations that might be written to when the function is called. To avoid such leakages the function is only allowed to write to memory cells with security levels greater than ℓ. Reference types $t \ \mathbf{ref}^{\ell}$, besides the security level t of the value stored at the associated location, carry a level ℓ which represents the security level of the reference itself. This is because references are themselves first-class values and can hence be used to leak confidential information.

As is common, a database is a collection of tables. Each table consists of at least one named column, each of which equipped with a fixed security type. The security levels on types for database columns express the confidentiality of the data contained in that column. In particular, each database is given a type signature Σ to express security policies for databases. A type signature describes tables as lists of records. Each record field corresponds to a column in the sense that the field name matches the name of the column in the database. The security level of a column is specified by using a suitable type for the corresponding field in the record. The ordering of elements in a list is irrelevant.

Types are equipped with a subtyping relation $\sqsubseteq$, which is an extension of the lattice ordering relation. The subtyping relation is standard [30, 24], therefore we do not report it here. With a little abuse of notation, we use the subtyping relation to compare security annotations ℓ with types t. In particular, if the type carries a security annotation ℓ', we compare the security levels $\ell \sqsubseteq \ell'$. Otherwise, we need to open the type and look inside the type constructor as described in Figure 2.

$$\frac{\ell \sqsubseteq \ell'}{\ell \sqsubseteq t^{\ell'}} \qquad \frac{}{\ell \sqsubseteq \mathbf{unit}} \qquad \frac{\ell \sqsubseteq pc \qquad \ell \sqsubseteq t}{\ell \sqsubseteq t' \overset{pc}{\to} t}$$

$$\frac{\ell \sqsubseteq t_1 \qquad \ell \sqsubseteq t_2}{\ell \sqsubseteq t * t} \qquad \frac{\ell \sqsubseteq t_i}{\ell \sqsubseteq \{\overline{f : t}\}} \qquad \frac{\ell \sqsubseteq t}{\ell \sqsubseteq \mathbf{Expr}\langle t \rangle}$$

Figure 2: Security annotation constraints

To illustrate the addition of security levels to the type system in the case of multi-tier applications, consider an example involving a database of people locations and friends, LocationDB. The locations are confidential, while the names are not, which leads to the following type for LocationDB.

```
LocationDB :
  { People :
    { Id : int^L; Name : string^L;
      Lon : float^H; Lat : float^H } list^L
  ; Friends :
    { Id1 : int^L ; Id2 : int^L } list^L
  }
```

Suppose John wants to know whether there are any friends within the range of 1km from his current location. We can query the database for the list of John's friends and later calculate the distance relative to John's location. This can be done by iterating once over all friends in the database to retrieve the list of John's friends and twice over all people in the database to retrieve the result information. After finding John's Id in the database, we check that whenever it occurs in the Friends table as Id1, the corresponding friend as Id2 occurs in the People table as Id. In that case, the name, the latitude and the longitude of that friend is returned as part of the result.

```
let db = <@ database "LocationDB" @>
type ResultType={name:string^L; lon:float^H; lat:float^H}

let friendsLoc : Expr < ResultType list^L > =
  <@ for f in (% db).Friends do
     for p1 in (% db).People do
     for p2 in (% db).People do
     if (p1.Name = "John") &&  (p1.Id = f.Id1) &&
        (f.Id2 = p2.Id) then
     yield ({name = p2.Name; lon = p2.Lon; lat = p2.Lat})
  @>
```

The information flow policy for the program is specified by giving a type annotation to the quoted expression that generates the query, i.e., a type annotation for friendsLoc. In particular the name component of the result is public, while the location information is confidential as described by ResultType. This matches the policy specified for the database contents, i.e., LocationDB, in which the name of people are public while their locations are not. Changing the security annotation of the name field from public to confidential should result in a type error, since the security level of the Name field of the result is public. The example so far illustrates secure information flows from the database to the server for an attacker model where the server is untrusted.

The server uses the result of the database query to calculate the distance between John's location and his friends location, and then send to John the list of nearby friends. The function dist : $(float^{\ell} * float^{\ell}) * (float^{\ell} * float^{\ell}) \xrightarrow{\ell'} float^{\ell}$ is side-effect free and it computes the Euclidean distance between two points. The security annotations are parametric on the security levels of inputs and outputs.

```
let friendNames : float^L * float^L -> string^L list^L  =
  <@ fun  publicLoc ->
     let res = run friendsLoc in
     for r in res do
     if dist((r.lon, r.lat), publicLoc) <= 1 then
     yield ({ name = r.name}) @>
```

The function friendNames takes as input a public location publicLoc, executes the query represented by the function friendsLoc on the database and returns a list of public names of nearby friends. Since the location information contained in the result of friendsLoc is confidential, there is an implicit flow from the location to the list of names. In fact, a public observer learns that the location of everyone in the returned list of names is within 1km from the location publicLoc. Therefore, the security type checking should fail. However, one may consider acceptable to leak the distance information as long as the exact location is protected. This can be achieved by *declassifying* the function dist, i.e., considering its result as public, although part of the input is confidential. At last, John can call the remote function friendNames on the client-side by providing his current location locJohn.

```
let locJohn : (float^L, float^L) = GetLocation()

let friends : string^L list^L = friendNames locJohn
```

The function is executed on the server-side and it interacts with the database to retrieve information as described above. Then the list of names of nearby friends is returned back to John on the client-side. The security type checker will ensure that there are no insecure information flows, except the allowed ones, from the database to the client.

2.2 Operational Semantics

The operational semantics of the language evaluates terms in the context of a mutable store μ and a database Ω. A partial mapping $\mu : Loc \to Val$ from locations to values models the semantics of memory effects. We write $\mu[l \mapsto v]$ for a store μ which maps location l to value v, otherwise agrees with μ. A *configuration* (e, μ) is a pair of a term e and a store μ. We write e when μ is empty. We denote evaluation of a configuration (e, μ) using database data in Ω to another configuration (e', μ') by $(e, \mu) \longrightarrow_{\Omega} (e', \mu')$. Ω is a function that maps database names to the actual content of the database it refers to, and δ is a function that maps operators to their corresponding semantics. Σ maps constants and databases to their respective types. We assume that Ω is consistent with the typing for databases given in Σ: for each database $\Omega(db)$ is assumed to be a value of type $\Sigma(db)$. Let $\longrightarrow_{\Omega}^{*}$ be the reflexive-transitive closure of $\longrightarrow_{\Omega}$. Evaluation and normalization of the quoted language is denoted by $eval_{\Omega}(norm(e))$. Figure 8 shows the syntax of normalized terms. This evaluation generates database queries that can be translated to SQL and executed by actual database servers. For instance, higher-order features such as nested records or function applications need to be evaluated to obtain computations that can be expressed in SQL. The syntax of values and evaluation contexts can be defined both for the host language and the quoted language as described in Figure 9. The quoted language is purely functional and contains no recursion. The evaluation contexts ensure that the semantics is call-by-value with left-to-right evaluation of terms. Quotation contexts $\mathcal{Q}$ are used to ensure that there are no antiquotations left of the hole. The evaluation rules for the host language are standard as reported in Figure 10. We denote the substitution of free occurrences of variable x in term e with another term e' by $e[x \mapsto e']$. The evaluation contexts entail sequentiality and let-binding between terms; we write $e_1; e_2$ for $(\mathbf{fun}(x) \to e_2)e_1$, where x is not free in e_2 and $\mathbf{let}\ x = e\ \mathbf{in}\ e'$ for $(\mathbf{fun}(x) \to e')\ e)$. The evaluation rules for the query language, as presented in Figures 11 and 12, follow Cheney et al. [14]. To avoid clutter, we omit the store component from configurations since the quoted language is purely functional.

2.3 Security Condition

The security condition expresses the notion of noninterference for a functional language with references and databases. Noninterference is an information flow policy that formalizes computational independence between confidential and public information, guaranteeing that no information about the former can be inferred from the latter. More precisely, this is expressed as the preservation of an equivalence relation under pairwise execution; given two inputs that are equal in the components that are visible to an attacker, evaluation should result in two output values that also coincide in the components that can be observed by the attacker. Memory locations are not directly observable by the attacker, however their contents may affect the output returned by the computations and thus leak information. For example, the program $\textbf{let } l = \textbf{ ref true}^{\mathtt{H}} \textbf{ in } !l$ uses a public location l, which stores a confidential value **true**, to leak that value to an attacker through the dereference $!l$.

To establish the behavior of a secure program from the perspective of an attacker, we introduce the notion of low-equivalence denoted by $\sim$ that demands that parts of values with types that are annotated with `L` are equal, while placing no demands on the high counterparts. Low-equivalence is formalized as a family of equivalence relations $\sim_t$ on values parametrized by types. We omit the subscript on $\sim$ when the type is clear from the context and write $\sim$ for sequences of values. To present the relations in a more concise manner, we combine the cases for different security levels using implication in the premises; e.g. equality on base types is only required if the security level is `L`.

Definition 1 ($\sim_t$)**.** *The family of equivalence relations $\sim_t$ is defined inductively by the rules in Figure 3.*

$$\frac{\ell = \mathtt{L} \Rightarrow c' = c''}{c' \sim_{c^{\ell}} c''} \qquad \frac{\begin{array}{c}\forall v_1, v_2, v_1', v_2', \Omega_1, \Omega_2.\\ (\Omega_1 \sim_{\Sigma} \Omega_2 \wedge v_1 \sim_t v_2 \wedge\\ e_1[x \mapsto v_1] \longrightarrow^{*}_{\Omega_1} v_1' \wedge\\ e_2[x \mapsto v_2] \longrightarrow^{*}_{\Omega_2} v_2') \Rightarrow\\ v_1' \sim_{t'} v_2' \wedge pc \sqsubseteq t'\end{array}}{\textbf{fun}(x) \to e_1 \sim_{t \xrightarrow{pc} t'} \textbf{fun}(x) \to e_2}$$

$$\frac{\ell = \mathtt{L} \Rightarrow \;\; l_1 = l_2}{l_1 \sim_{t\ \textbf{ref}^{\ell}} \;\; l_2} \qquad \frac{\ell = \mathtt{L} \Rightarrow (|[\overline{v}]| = |[\overline{w}]| \wedge \overline{v \sim_t w})}{[\overline{v}] \sim_{(t\ \textbf{list})^{\ell}} [\overline{w}]}$$

$$\frac{\begin{array}{c}\forall v_1, v_2, v_1', v_2', \Omega_1, \Omega_2.\\ \Omega_1 \sim_{\Sigma} \Omega_2 \wedge v_1 \sim_t v_2 \wedge\\ e_1[f \mapsto \textbf{rec}\ f(x) \to e_1, x \mapsto v_1] \longrightarrow^{*}_{\Omega_1} v_1' \wedge\\ e_2[f \mapsto \textbf{rec}\ f(x) \to e_2, x \mapsto v_2] \longrightarrow^{*}_{\Omega_2} v_2' \Rightarrow\\ v_1' \sim_{t'} v_2' \wedge pc \sqsubseteq t'\end{array}}{\textbf{rec}\ f(x) \to e_1 \sim_{t \xrightarrow{pc} t'} \textbf{rec}\ f(x) \to e_2}$$

$$\frac{\overline{v \sim_t w}}{\{\overline{f = v}\} \sim_{\{\overline{f:t}\}} \{\overline{f = w}\}} \qquad \frac{v_1 \sim_{t_1} v_1' \qquad v_2 \sim_{t_2} v_2'}{(v_1, v_2) \sim_{t_1 * t_2} (v_1', v_2')}$$

$$\frac{}{() \sim_{\textbf{unit}} ()} \qquad \frac{\begin{array}{c}\forall \Omega_1, \Omega_2. \Omega_1 \sim \Omega_2 \Rightarrow\\ eval_{\Omega_1}(norm(e_1)) \sim_t eval_{\Omega_2}(norm(e_2))\end{array}}{e_1 \sim_{\textbf{Expr}\langle t \rangle} e_2}$$

Figure 3: Rules for $\sim_t$

Built-in values c of base type b are compared using equality if the values are public. Unit values $()$ are related by $\sim_{\textbf{unit}}$ and do not contain security levels. In the case of function types and quoted expressions, $\sim_t$ corresponds to noninterference for the bodies of the functions. Moreover, functions are related by $\sim_{t \xrightarrow{\ell} t'}$ if for all input values related by $\sim_t$ they evaluate to values related by $\sim_{t'}$ and the memory effects are upper bounded by the security level of the result $\ell \sqsubseteq t'$. Records are related by $\sim$ if they contain the same fields, and each field's contents are also related by $\sim$. Similarly, tuples are related by $\sim$ if the corresponding components are related by $\sim$. Two lists are required to have the same length if the list type is annotated with `L`, but their contents may differ based on the element type. To illustrate this, consider two lists of integers $l_1 = \textbf{yield}\ 1\ @\ []$ and $l_2 = \textbf{yield}\ 2\ @\ []$. If the lists are typed with the type $t = (\textbf{int}^{\mathtt{H}}\ \textbf{list})^{\mathtt{L}}$, the length of the list is considered public, while the contents are confidential. If in contrast the type is $t' = (\textbf{int}^{\mathtt{L}}\ \textbf{list})^{\mathtt{L}}$, neither the contents nor the length of the list is confidential. Hence $l_1 \sim_t l_2$ holds while $l_1 \sim_{t'} l_2$ does not. Memory locations are compared using equality if the locations are public.

With this we are ready to define the top-level notion of security based on *noninterference* [20]. Since the family of low-equivalence relations is parametrized by types the definition is done with respect to the initial host type, the initial database type and the final result type.

Definition 2 ($NI(e_1, e_2)_{t,\Sigma,t'}$)**.** *Two expressions e_1 and e_2 are noninterfering with respect to the host type t, the database type Σ and the final type t' if for all Ω_i, v_i, v_i' and μ_i such that $v_1 \sim_t v_2$, $\Omega_1 \sim_{\Sigma} \Omega_2$, and $e_i[x \mapsto v_i] \longrightarrow^{*}_{\Omega_i} (v_i', \mu_i)$ for $i \in \{1, 2\}$ it holds that $v_1' \sim_{t'} v_2'$.*

Given an open expression e, $NI(e, e)_{t,\Sigma,t'}$ should be read as e is secure with respect to the security policy expressed by t, Σ and t', i.e., no secret parts of host and the database as defined, respectively, by t and Σ is able to influence the public parts of the result value as defined by t'. Note that the definition can represent expressions with multiple inputs by using record values. Moreover, the noninterference policy is *termination-insensitive* [41, 34], namely it ignores leaks via the observation of (non)termination.

Declassification: Noninterference is overly restrictive for programs that leak confidential information in a controlled manner, as shown by the example in Section 2.1. To account for these cases, we extend the framework with support for declassification policies that regulate what information can be released by the program. The policies are expressed in terms of *escape hatches* from a set $\mathcal{D} = \{d_1, \cdots, d_k\}$ and correspond to the *What* dimension in [35]. Escape hatches were introduced to express a similar notion, called *delimited release*, for imperative languages [33]. The security condition is then refined to also take into account the equivalence between declassification expressions. This requires to extend the low-equivalence relations used for noninterference with declassification.

Definition 3 ($DNI(e_1, e_2)_{\mathcal{D},t,\Sigma,t'}$)**.** *Two expressions e_1 and e_2 are noninterfering with respect to the declassification expressions $\mathcal{D}$, the host type t, the database type Σ and the final type t' if for all Ω_i, v_i, v_i' and μ_i such that $v_1 \sim_t v_2$, $\Omega_1 \sim_{\Sigma} \Omega_2$, $d_j[x \mapsto v_1] \sim_{t,\Sigma} d_j[x \mapsto v_2]$ and $e_i \longrightarrow^{*}_{\Omega_i} (v_i, \mu_i)$ for $i \in \{1, 2\}$ it holds that $v_1' \sim_{t'} v_2'$.*

2.4 Security Type System

The goal of the security type system is to enforce the notion of noninterference for a functional language with references and databases. We present the typing rules for the host language in Figure 4. Typing judgments are of the form $pc, \Gamma, M \vdash e : t$ where pc is the program counter level, Γ is a typing context mapping variables to types, M is a typing context mapping locations to types, e is an expression and t is a type. They denote that expression e has type t in context pc, Γ, M. We also write H for pc, Γ, M. Intuitively, the program counter level approximates the information that can be learned by observing that the program has reached a particular point during the execution and it is used to control implicit flows due to branching on high values. For uniformity, we write $pc, \Gamma, M \vdash v : t$ for typing judgments dealing with values, although pc is redundant given that values have no computational effects. $\ell \sqcup \ell'$ denotes the join of levels ℓ and ℓ', i.e., $\ell \sqcup \ell' = \mathtt{H}$ iff $\mathtt{H} \in \{\ell, \ell'\}$, and $\ell \sqcup \ell' = \mathtt{L}$ otherwise.

The typing rules for the quoted language are similar to those for the host language and are reported in Figure 5. Typing judgments have the form $H, \Delta \vdash e : t$, where H is the typing context for the host language and Δ is the typing context for the quoted language. We use the suffix Q to refer to the rule for the quoted language.

Most types contain a level ℓ that denotes whether the "structure" of the value is confidential. In the case of base types, this means that their values are confidential or not. In the case of $(t\ \mathbf{list})^{\ell}$, the level ℓ indicates whether the length of the list is confidential. If $\ell = \mathtt{H}$, the entire list is considered a secret, otherwise the length of the list may be disclosed to a public observer. However, the elements of the list may or may not be confidential depending on the level of the elements given by the type t. Record types, pair types and quotation types do not carry an explicit level annotation, since their security level is contained in the type components. In the case of records and pairs, it suffices to annotate the type of each component, since the structure can not be modified dynamically. For types for quoted expressions, the security annotation is contained in the type t. Function types contain the usual input and output types together with a security level pc which represents a lower bound on the security level of locations that may be written when calling the function. In order to securely call the function in a context pc' it must be the case that $pc' \sqsubseteq pc$. The intuition is that, in the presence of side-effects, the function can disclose information via its result or via its side-effects. We assume that types for operators, constants, and databases are given by the mapping Σ. Moreover, we also assume that each query only uses a single database. Expressions in the host language differ from expressions in the quoted language. Recursion, quotation, branching (rule If) and memory operations (reference creation, dereferencing and update) are only allowed in the host language; expressions of the form $\mathbf{database}(x)$ and antiquotations are only allowed in the quoted language.

ConstQ

$$\frac{\Sigma(c) = t}{H, \Delta \vdash c : t^{\ell}}$$

FunQ

$$\frac{H, \Delta, x : t \vdash e : t'}{H, \Delta \vdash \mathbf{fun}(x) \to e : t \to t'}$$

VarQ

$$\frac{x : t \in \Delta}{H, \Delta \vdash x : t}$$

ApplyQ

$$\frac{H, \Delta \vdash e_1 : t \to t' \qquad H, \Delta \vdash e_2 : t}{H, \Delta \vdash e_1\ e_2 : t'}$$

OpQ

$$\frac{\Sigma(op) = \overline{t} \to t \qquad \overline{H, \Delta \vdash M : t^{\ell}}}{H, \Delta \vdash op(\overline{M}) : t^{\bigsqcup \ell_i}}$$

Antiquote

$$\frac{H \vdash e : \mathbf{Expr}\langle t \rangle}{H, \Delta \vdash (\%\ e\ ) : t}$$

PairQ

$$\frac{H, \Delta \vdash e_1 : t_1 \qquad H, \Delta \vdash e_2 : t_2}{H, \Delta \vdash (e_1, e_2) : t_1 * t_2}$$

FstQ

$$\frac{H, \Delta \vdash e : t_1 * t_2}{H, \Delta \vdash \mathbf{fst}\ e : t_1}$$

SndQ

$$\frac{H, \Delta \vdash e : t_1 * t_2}{H, \Delta \vdash \mathbf{snd}\ e : t_2}$$

RecordQ

$$\frac{\overline{H, \Delta \vdash M : t}}{H, \Delta \vdash \{\overline{f = M}\} : \{\overline{f : t}\}}$$

ProjectQ

$$\frac{H, \Delta \vdash L : \{\overline{f : t}\}}{H, \Delta \vdash L.f_i : t_i}$$

YieldQ

$$\frac{H, \Delta \vdash M : t}{H, \Delta \vdash \mathbf{yield}\ M : (t\ \mathbf{list})^{\ell}}$$

NilQ

$$\frac{}{H, \Delta \vdash [] : (t\ \mathbf{list})^{\ell}}$$

ExistsQ

$$\frac{H, \Delta \vdash M : (t\ \mathbf{list})^{\ell}}{H, \Delta \vdash \mathbf{exists}\ M : \mathbf{bool}^{\ell}}$$

IfQ

$$\frac{H, \Delta \vdash L : \mathbf{bool}^{\ell} \qquad H, \Delta \vdash M : (t\ \mathbf{list})^{\ell'}}{H, \Delta \vdash \mathbf{if}\ L\ \mathbf{then}\ M : (t\ \mathbf{list})^{\ell \sqcup \ell'}}$$

UnionQ

$$\frac{H, \Delta \vdash M : (t\ \mathbf{list})^{\ell} \qquad H, \Delta \vdash N : (t\ \mathbf{list})^{\ell'}}{H, \Delta \vdash N\ @\ M : (t\ \mathbf{list})^{\ell \sqcup \ell'}}$$

ForQ

$$\frac{H, \Delta \vdash M : (t\ \mathbf{list})^{\ell} \qquad H, \Delta, x : t \vdash N : (t'\ \mathbf{list})^{\ell'}}{H, \Delta \vdash \mathbf{for}\ x\ \mathbf{in}\ M\ \mathbf{do}\ N : (t'\ \mathbf{list})^{\ell \sqcup \ell'}}$$

SubQ

$$\frac{t \sqsubseteq t' \qquad H, \Delta \vdash M : t}{H, \Delta \vdash M : t'}$$

DatabaseQ

$$\frac{\Sigma(db) = \{\overline{f : t}\}}{H, \Delta \vdash \mathbf{database}(db) : \{\overline{f : t}\}}$$

Figure 5: Typing rules for quoted language

We now comment on a few typing rules. Rules Var, VarQ and Loc assign types to variables and locations by looking up the corresponding environment. Fun and Rec use the program counter level appearing in the functions type to check the respective function bodies. Apply is used to check function application. The rule ensures that the side-effects pc' of the caller function are not visible in contexts for which the program counter level is pc, namely $pc \sqsubseteq pc'$. As a result, it prevents a function to write to low memory locations in a high context and thus leak information through implicit flows. Ref checks memory allocation operations. It ensures that a low reference is not created in a high context and that it does not contain a high value. Deref checks dereference operations and ensures that the reference level is upper bounded by the level of its contents to avoid information leakage through aliases. Assn checks memory updates and ensures that no low memory writes occur in a high context or in a high location. The following example captures the intuition behind the typing rules for mutable storage. Let `l, l'` be variables of type $\mathbf{int}^{\mathtt{L}}\ \mathbf{ref}^{\mathtt{H}}$, `l''` of type $\mathbf{int}^{\mathtt{H}}\ \mathbf{ref}^{\mathtt{H}}$

$$\textsc{Const}\quad \frac{\Sigma(c) = t}{pc, \Gamma, M \vdash c : t^{\ell}} \qquad \textsc{Unit}\quad \frac{}{pc, \Gamma, M \vdash () : \mathbf{unit}} \qquad \textsc{Var}\quad \frac{x : t \in \Gamma}{pc, \Gamma, M \vdash x : t} \qquad \textsc{Loc}\quad \frac{l : t \in M}{pc, \Gamma, M \vdash l : t} \qquad \textsc{Nil}\quad \frac{}{pc, \Gamma, M \vdash [] : (t\ \mathbf{list})^{\ell}}$$

$$\textsc{Fun}\quad \frac{pc, \Gamma, x : t, M \vdash e : t'}{pc', \Gamma, M \vdash \mathbf{fun}(x) \to e : (t \overset{pc}{\to} t')} \qquad \textsc{Rec}\quad \frac{pc, \Gamma, x : t, f : t \overset{pc}{\to} t', M \vdash e : t'}{pc', \Gamma, M \vdash \mathbf{rec}\ f(x) \to e : t \overset{pc}{\to} t'} \qquad \textsc{Lift}\quad \frac{pc, \Gamma, M \vdash e : t}{pc, \Gamma, M \vdash \mathbf{lift}\ e : \mathbf{Expr}\langle t \rangle}$$

$$\textsc{Exists}\quad \frac{pc, \Gamma, M \vdash e : (t\ \mathbf{list})^{\ell}}{pc, \Gamma, M \vdash \mathbf{exists}\ e : \mathbf{bool}^{\ell}} \qquad \textsc{Op}\quad \frac{\Sigma(op) = \bar{t} \to t \qquad \overline{pc, \Gamma, M \vdash e : t^{\ell}}}{pc, \Gamma, M \vdash op(\overline{e}) : t^{\bigsqcup \ell_i}} \qquad \textsc{Yield}\quad \frac{pc, \Gamma, M \vdash e : t}{pc, \Gamma, M \vdash \mathbf{yield}\ e : (t\ \mathbf{list})^{\ell}}$$

$$\textsc{Apply}\quad \frac{pc, \Gamma, M \vdash e_1 : t \overset{pc'}{\to} t' \qquad pc, \Gamma, M \vdash e_2 : t \qquad pc \sqsubseteq pc'}{pc, \Gamma, M \vdash e_1\ e_2 : t'} \qquad \textsc{Pair}\quad \frac{pc, \Gamma, M \vdash e_1 : t_1 \qquad pc, \Gamma, M \vdash e_2 : t_2}{pc, \Gamma, M \vdash (e_1, e_2) : t_1 * t_2}$$

$$\textsc{Fst}\quad \frac{pc, \Gamma, M \vdash e : t_1 * t_2}{pc, \Gamma, M \vdash \mathbf{fst}\ e : t_1} \qquad \textsc{Snd}\quad \frac{pc, \Gamma, M \vdash e : t_1 * t_2}{pc, \Gamma, M \vdash \mathbf{snd}\ e : t_2} \qquad \textsc{Record}\quad \frac{\overline{pc, \Gamma, M \vdash e : t}}{pc, \Gamma, M \vdash \{\overline{f = e}\} : \{\overline{f : t}\}} \qquad \textsc{Project}\quad \frac{pc, \Gamma, M \vdash e : \{\overline{f : t}\}}{pc, \Gamma, M \vdash e.f_i : t_i}$$

$$\textsc{Union}\quad \frac{pc, \Gamma, M \vdash e : (t\ \mathbf{list})^{\ell} \qquad pc, \Gamma, M \vdash e' : (t\ \mathbf{list})^{\ell'}}{pc, \Gamma, M \vdash e'\ @\ e : (t\ \mathbf{list})^{\ell \sqcup \ell'}} \qquad \textsc{For}\quad \frac{pc, \Gamma, M \vdash e : (t\ \mathbf{list})^{\ell} \qquad pc, \Gamma, x : t, M \vdash e' : (t'\ \mathbf{list})^{\ell'}}{pc, \Gamma, M \vdash \mathbf{for}\ x\ \mathbf{in}\ e\ \mathbf{do}\ e' : (t'\ \mathbf{list})^{\ell \sqcup \ell'}}$$

$$\textsc{If1}\quad \frac{pc, \Gamma, M \vdash e : \mathbf{bool}^{\ell} \qquad pc, \Gamma, M \vdash e' : (t\ \mathbf{list})^{\ell'}}{pc, \Gamma, M \vdash \mathbf{if}\ e\ \mathbf{then}\ e' : (t\ \mathbf{list})^{\ell \sqcup \ell'}} \qquad \textsc{If}\quad \frac{pc, \Gamma, M \vdash e : \mathbf{bool}^{\ell} \qquad pc \sqcup \ell, \Gamma, M \vdash e_i : t \qquad \ell \sqsubseteq t \qquad i \in \{1, 2\}}{pc, \Gamma, M \vdash \mathbf{if}\ e\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2 : t}$$

$$\textsc{Deref}\quad \frac{pc, \Gamma, M \vdash e : t\ \mathbf{ref}^{\ell} \qquad \ell \sqsubseteq t}{pc, \Gamma, M \vdash !e : t} \qquad \textsc{Quote}\quad \frac{pc, \Gamma, M, \cdot \vdash e : t}{pc, \Gamma, M \vdash \texttt{<@}\ e\ \texttt{@>} : \mathbf{Expr}\langle t \rangle} \qquad \textsc{Sub}\quad \frac{t \sqsubseteq t' \qquad pc, \Gamma, M \vdash e : t}{pc, \Gamma, M \vdash e : t'}$$

$$\textsc{Run}\quad \frac{pc, \Gamma, M \vdash e : \mathbf{Expr}\langle t \rangle}{pc, \Gamma, M \vdash \mathbf{run}\ e : t} \qquad \textsc{Ref}\quad \frac{pc, \Gamma, M \vdash e : t \qquad pc \sqsubseteq t}{pc, \Gamma, M \vdash\ \mathbf{ref}\ e : t\ \mathbf{ref}^{pc}} \qquad \textsc{Assn}\quad \frac{pc, \Gamma, M \vdash e_1 : t\ \mathbf{ref}^{\ell} \qquad pc, \Gamma, M \vdash e_2 : t \qquad pc \sqcup \ell \sqsubseteq t}{pc, \Gamma, M \vdash e_1 := e_2 : \mathbf{unit}}$$

Figure 4: Type system for host language

and `h` of type $\mathbf{bool}^{\mathtt{H}}$. The program is insecure since the returned value at location `l` reveals the initial value of variable `h` through aliasing.

```
l = ref 0; l' = ref 1; let  l'' =
if h then l else l' in l'':= 2; !l
```

The program is correctly rejected by the type system. By rule Ref the first two references are typable for $pc = \mathtt{H}$. The conditional is also typable by rule If, since `l` and `l'` are high references. The successive assignment is typable by rule Assn provided that 2 has type $\mathbf{int}^{\mathtt{H}}$. The type checking fails when considering the dereference $!l$, since the rule Deref requires $\ell \sqsubseteq t$, which is not true for `l` of type $\mathbf{int}^{\mathtt{L}}\ \mathbf{ref}^{\mathtt{H}}$.

Lists can be assigned an arbitrary level when constructed using **yield** and []. Expressions of the form e_1 @ e_2 reveal information about the structure of both lists and hence their security levels are combined in the result type. Similarly, **exists** only reveals information about the structure of the list, but nothing about the contents. Therefore, the security level of list contents is discarded and only the security level of the list itself is present in the result type. Rule Quote ensures that its arguments are typed in an empty context for quoted expressions. This expresses that only closed quoted terms are allowed in this language. Running a quoted expression e of type $\mathbf{Expr}\langle t \rangle$ using **run** e results in an expression of type t (rule Run). Expressions for **database**(db) get their type from the mapping Σ. Rule Antiquote allows to entities defined in the host language from within a quoted expression. The argument of an antiquotation must itself be a quoted expression. Rules Sub and SubQ allows raising the security level of an expression.

To illustrate the type system further, we explain the typing rule For rule in greater detail. Recall that **for** expressions are used to denote list comprehensions. The typing rule assigns the resulting list the join of the security level of both sub-expressions. The following example demonstrates why this is required.

Consider the program **for** x **in** xs **do** ys that uses a **for** expression to leak the structure of the lists xs and ys. We assume xs to have type $(t\ \mathbf{list})^{\ell}$ for some type t and level ℓ, whereas ys has type $(t'\ \textbf{list})^{\ell'}$. Since the resulting lists for each element of xs will be concatenated, the resulting list will have length $|xs| \times |ys|$, where $|a|$ denotes the length of a. If either xs or ys contains only one element, the length of the other list is revealed through the result. To account for this information flow, the resulting list will be typed with level $\ell \sqcup \ell'$.

2.5 Soundness

The soundness result is stated as the preservation of a low-equivalence relation under pairwise execution. If we start out in any two low-equivalent environments then the result of running a well-typed program will be low-equivalent with respect to the type of the program. Assuming that the typing of the execution environment corresponds to the capabilities of the attacker, noninterference guarantees that all information observable by the attacker is independent of confidential information. To make the connection between the host policy Γ, the database policy Σ and the type system explicit we write $\Gamma, \Sigma \vdash e : t$ even though Σ was kept implicit in the typing rules.

Theorem 1 (Soundness). *If $x : t, \Sigma \vdash e : t'$, then $NI(e,e)_{t,\Sigma,t'}$.*

Proof sketch. The theorem is proved by adapting the proof technique introduced by by Pottier and Simonet [38] for an ML-like security-typed language. This is done by defining an extension of the language which allows reasoning about pairs of program configurations, and then showing that the type system for the extended language enjoys the subject reduction property. Then noninterference follows as a result of the subject reduction theorem. The proof can be found in the appendix. □

The type system for the host language and the quoted language can be extended with two additional rules which take into account declassification through expressions from the set $\mathcal{D}$. Intuitively, the rules allow to downgrade the security level of an expression if that expression is in the set of declassified expressions $\mathcal{D}$ and the level *pc* is upper bounded by the level of the declassified expression. The latter is used to enforce that no sensitive information is released implicitly through the declassification mechanism.

$$\textsc{Decl}\quad \frac{pc, \Gamma, M, \mathcal{D} \vdash d : t \qquad pc \sqsubseteq t \qquad (d, t') \in \mathcal{D}}{pc, \Gamma, M \vdash d : t'}$$

$$\textsc{DeclQ}\quad \frac{H, \Delta, \mathcal{D} \vdash d : t \qquad pc \sqsubseteq t \qquad (d, t') \in \mathcal{D}}{H, \Delta \vdash d : t'}$$

Theorem 2 (Soundness under Declassification). *If $x : t, \Sigma, \mathcal{D} \vdash e : t'$, then $DNI(e,e)_{\mathcal{D},t,\Sigma,t'}$.*

3. JSLINQ

Figure 6 shows the architecture of JSLINQ. The input is an F# project consisting of the security policy and the application code. The right branch of the figure shows how a project is first compiled to a 3-tier application using the unmodified build process for web applications based on WebSharper. The code of the project is used to create a 3-tier application consisting of JavaScript created using WebSharper, .NET assemblies for server-side logic and SQL queries for the database, created using LINQ. Upon successful compilation, JSLINQ's security type checker can be used on the F# project to determine if the application complies to the specified information-flow policy. How the resulting 3-tier application and the verification result are used depends on the use case of JSLINQ: one possibility is to discard non-compliant application builds and to deploy compliant applications into production. The remainder of the section discusses JSLINQ components in more detail.

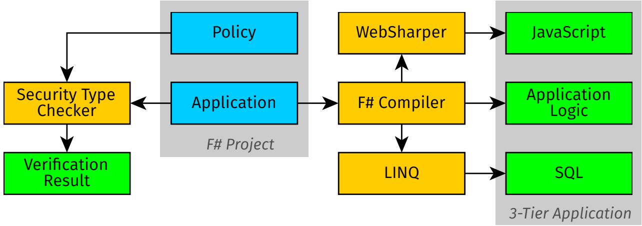


Figure 6: JSLINQ Architecture

WebSharper: WebSharper is a fully-featured and commercially supported framework for web application development in F#, providing powerful functional abstractions such as sitelets for document definition, formlets for data entry forms and flowlets for workflows [21]. Moreover, it offers abstractions for essential web concepts such as the DOM or JavaScript code. Importantly, these abstractions enjoy type safety properties, allowing to leverage the F# type system to build robust applications. One of WebSharper's key features is the translation of F# functions into JavaScript code for execution in the browser. Server-side functions can be designated as remote procedure calls (RPC), and can be transparently called in client-side code, as in the example:

```
// Server-side function called by the client via AJAX.
[<Remote>]
let getText () = "JSLINQ"
// Client-side function translated to JavaScript and HTML.
[<JavaScript>]
let Main () = Text (getText ())
```

WebSharper supports extensions of the client with third-party libraries, for example a map service. Third-party libraries usually consist of JavaScript code that is embedded into the page. Calls from the client-side F# code to the embedded third-party library are handled by wrappers that provide an F# interface to the JavaScript code. This approach requires full trust on the JavaScript code provided by the third party. However, JSLINQ can be used to type-check third-party libraries written in F#. This allows rewriting crucial third-party JavaScript libraries in F# to make them amenable to security analysis using JSLINQ.

F# Project: JSLINQ is designed to perform the verification step after successful compilation of the project. JSLINQ processes MSBuild projects and it is integrated with Microsoft Visual Studio. Code within a project is either part of the policy or part of the program. The policy controls information flows via security type signatures which are added to the definitions of functions and databases. The program implements the application and is subject to the security type check according to the policy. Since the policy is expressed within normal F# syntax, the use of JSLINQ does not interfere with the normal build process of the application and the use of standard tools.

Policy: The policy is specified by adding custom attributes with security type signatures to declarations. Signatures are represented as strings that follow the language in Section 2.1, and use variables for security levels in order to support polymorphism. If no security level is specified within a signature, the corresponding level variable is unconstrained. The following code fragment demonstrates how signatures are added to F# declarations:

```
[<SecT("_^H")>]
let boolH = true

[<SecT("unit ->^L  _^L")>]
let f () = 1
```

We divide a web-application policy into three types: a library policy, an RPC policy and a database policy. Each type deals with different tiers and the meaning of a security type signature depends on the tier in which it is located.

The policy for library functions is defined in a separate module, which is marked with a policy attribute. All library functions used by the program need to be wrapped in the policy, otherwise their use is not allowed. Since HTML and JavaScript abstractions of WebSharper are also library functions, the policy for client-side functionality is specified in this part. Each wrapper function has a mandatory security type signature that governs which security levels are used when the wrapper is called. The following snippet demonstrates a wrapper that uses WebSharper functions to generate a masked input field for passwords, labelled as high:

```
[<Policy>]
module Policy =
  [<JavaScript>][<SecT("unit -> _^H")>]
  let InputPW () = Input [Attr.Type "password"]
```

The policy for RPCs from the client to the server consists of attributes to the declarations of RPC functions within the program. We define the RPC policy and the program in the same file for sake of simplicity. However, JSLINQ allows a complete separation of policy and program into separate files, as we do for the other parts of the policy. Type signatures on RPC functions restrict the information flow from the client to the server (via function arguments) and from the server to the client (via return values). The following fragment demonstrates flows in both directions:

```
[<Remote>][<SecT("unit -> _^L")>]
let untrustedClient () = true

[<Remote>][<SecT("_^L ->^L unit")>]
let untrustedServer (x:bool) = ()
```

The database policy is defined by adding security type signatures to an attribute-based mapping for LINQ [3]. Security type signatures are added to table and column definitions as shown in the following example:

```
[<Table>][<SecT("_^L")>] // Public table length
type Account =
  [<Column>][<SecT("_^L")>] // Public username
  abstract member Username : string
  [<Column>][<SecT("_^H")>] // Confidential password
  abstract member Password : string
```

Security Type Checker: The design of JSLINQ as a verification step after compilation allows us to assume that the code has correct syntax, data types and satisfied dependencies, hence the implementation can only focus on the security type check. Noteworthy, we leave the F# type system untouched and maintain a completely separate security type system during the verification. We perform the security type checking in two steps, which we repeat for each top-level declaration found in the code: first we recursively traverse AST for the declaration to obtain set of constraints and a security type signature by means of the FParsec library [6]. The second step substitutes level variables with actual security levels by solving the constraint set. The resulting types and possibly remaining constraints are added to the environment before proceeding with the next declaration. JSLINQ uses the AST generated by the F# compiler, which is retrieved using the library FSharp Compiler Services [5]. We thus do not duplicate compiler features that are unrelated to the security type check and benefit from F#'s desugaring. This is a clear advantage over prototypes, e.g. SELINQ or SIF, that enhance existing type systems.

4. CASE STUDIES

We have used JSLINQ to implement several case studies as F# projects. In this section we first describe the general design of the policy language and then remark on the policy requirements for the case studies that we have implemented.

4.1 Library Policy

The largest part of the library policy are the signatures for the DOM and JavaScript abstractions. The documents shown in the browser are constructed using these abstractions at runtime. For simplification, we consider the HTML elements as trusted sinks. The rationale behind this is that the user has full access to the data once it has arrived in the browser, independently of that data being displayed or not. However, this assumption does not hold for the full WebSharper API, as it would allow to write and read the elements in the DOM tree in various ways. Therefore, the policy only permits basic operations on the DOM. An important exception from our trusted sink assumption are HTML elements which load external resources, such as images and IFrames. These elements can be used to leak data either directly within the source attribute or indirectly via externally observable HTTP requests. Therefore, we annotate the creation of the source attribute with low security level, both for the URL argument and the side-effects.

4.2 Scenario Discussion

We now comment on different aspects of the policy and provide examples for vulnerabilities captured by JSLINQ.

Password Meter We have included the password meter to demonstrate a policy with full client isolation, where the password is not allowed to leave the browser. The policy declares password fields as sensitive sources. Leaks to third parties and to the application server are prevented by assigning low levels to the source attribute and to the arguments and side-effects of RPC functions, respectively. The scenario assumes that the server is untrusted, as it should not receive the password. A problem with this view is that the JavaScript code executed by the client is usually delivered by an untrusted server. This means that the integrity of the client-side code after the security type check is not guaranteed. Such changes are not subject to the security policy and can thus be abused to leak confidential data. Therefore we have to put trust in the integrity of the code delivered by the application server, which we summarize as *partial trust.* Alternatively, remote attestation methods such as code or certificate signatures can ensure code integrity. The following snippets show a secure password check and two leaks via the source attribute that are handled correctly by JSLINQ. The scenario consists of 53 F# and 6215 generated JS LOCs.

```
let content = // Allowed: Secret only in browser.
```

```
  if (containsLetters password)
  then Text "Passed" else Text "Failed"

let content' = // Blocked: Leak via source attribute.
  Image [Src ("http://example.com/img.png?" + password)]
// Blocked: Leak via side-effects.
let content'' = Src (if secret == "jSL!Nq42"
  then "http://example.com/true.jpg"
  else "http://example.com/false.jpg")
```

Location-Based Service This scenario demonstrates declassification of a client-side secret, in this case the user's position. Third parties and the application server can only receive declassified (obfuscated) coordinates. We define declassification as a function that adds a random offset to the position. The function is applied to the confidential latitude and longitude values. The real coordinates are isolated in the browser in the same way as for the password meter. We provide two variants of the location-based service to showcase two different attacker models. The first example embeds a map via an IFrame, where the position is an argument to the source attribute of the IFrame. The following snippet shows how the use of declassified coordinates is permitted, while real coordinates are blocked:

```
let iframeSrc = Src // Allowed: Obfuscated coordinate.
  "https://maps.example.com/?q=" +
  (string (randomize Lat)) + "," + (string (randomize Lon))

let iframeSrc' = Src // Blocked: Exact coordinate.
  "https://maps.example.com/?q=" +
  (string Lat) + "," + (string Lon)
```

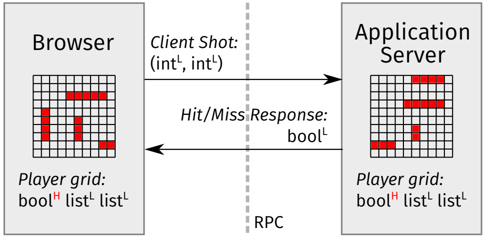
The second example includes a third-party library called via F#. We use the Google Maps extension for WebSharper and wrap the initialization and panning of the map within the policy, both having low side-effects and low values. Since the extension wraps the original JavaScript code, we have to fully trust the F#-to-JavaScript extension and JavaScript code implementing the WebSharper APIs. The scenario consists of 76 F# and 6279 generated JS LOCs.

Movie Rental This scenario demonstrates the use of security policies on databases. The database consists of a list of items (e.g. movies) subject to events (e.g. movie rentals) happening at a certain location and time. The location of an event is confidential, while all other information is public. The database policy assigns to the latitude and longitude high-security levels. Leaks to the client are prevented by labelling the return values of RPC functions as public. The following LINQ query joins rentals with movies and returns a list of movie titles. Movie titles are input to an RPC function which is only allowed to return public values. As a result the first `yield` statement is allowed to return the movie titles. If instead we use the second `yield` statement, JSLINQ rejects the program.

```
let events =  query {
    for e in db.Event do
    for i in db.Item do
    if e.ItemId = i.Id then
      (* Allowed *) yield i.Name
      (* Blocked *) yield (string e.Lat) }
```

Moreover, we allow the user to retrieve a ranking of popular movies within an area. The implementation contains a pre-defined set of areas which are addressed using indexes. The user can only specify the index for an area of interest. The application server filters the list of movie rentals based on the coordinate values. JSLINQ will infer a high-security level for the length of the resulting list, as it depends on the coordinate values. Our policy allows that geographic information about rentals is disclosed on the granularity of fixed-size areas, therefore we can directly declassify the length of the list. The scenario consists of 87 F# and 6231 generated JS LOCs.

Figure 7: Simplified IFC policy for Battleship



Friend Finder App In this scenario we consider a completely untrusted application server. The client obtains the code from a trusted source. We use the Apache Cordova framework [2] to package the client-side functionality as an app that can be distributed via a trusted channel. Cordova also provides access to the address book of the device. The app can access the address book only via a function defined in the policy, which assigns a high-security level to the contact details. The policy allows declassification by means of a hash function on strings. Leakage of plain contact details to the untrusted server is prevented by assigning a low security level to the arguments and side-effects of RPC functions. The following snippet illustrates a secure and an insecure RPC call:

```
// Allowed: Look-up of hashed phone number
let rpcResult = remoteLookup (Hash phoneNumber)
// Blocked: Look-up of plain phone number
let rpcResult' = remoteLookup phoneNumber
```

The scenario has 62 F# and 9966 generated JS LOCs.

Battleship We implement a simplified version of the classical Battleship game [29, 39]. The client uses the browser to play against the server and the goal of each player is to hide the exact position of their ships on a grid. Both sides trust each other to correctly follow the rules of the game, so we are only concerned about confidentiality. A desirable IFC policy for this game is to mark the values indicating individual ship positions as confidential and all parameters and return values of RPC functions as public, so that confidential information is not allowed to pass the barrier between the browser and the server. This allows us to re-use the same security policy on both sides, as shown in Figure 7. The game rules require declassification, since the response to a shot requires disclosure of one bit of information ("hit" or "miss") to the other player per round. On each side we have to perform declassification twice: firstly for the hit/miss response to a shot, as it directly depends on the presence of a ship at that location, and secondly for indicating to the opponent if a player is defeated, which requires to test all occupied cells. The latter can be done locally, but for implementation reasons players report their own defeat to the opponent. The following example shows this for the client-side:

```
let serverShotResult = {
  shot = response.shot;
  hit = DeclassifyBool !serverTarget.occupied;
  defeated = DeclassifyBool clientDefeated }
```

The scenario has 255 F# and 6348 generated JS LOCs.

4.3 Case Study Results

Table 1: Overview of implemented scenarios

	Trust			# of Annotations		
Scenario	Client	3rd Party	Server	API	RPC	DB
Password Meter	Yes	No	Partial	10	0	0
POI IFrame	Yes	No	Yes	10	1	5
POI Embedded	Yes	Yes	Yes	11	1	5
Movie Rental	No	No	Yes	9	1	8
Friend Finder	Yes	No	No	9	1	0
Battleship	Yes	No	Yes	12	4	0

Table 1 summarizes our case studies. The different combinations of client, third party and server trust illustrate the attacker models handled by JSLINQ. The initial effort of defining the API policy annotations comes with the benefit of minor burden on application programmer side. The policy for JSLINQ requires only very few annotations within the application code. As reported above, the LOCs for F# and JavaScript refer to the application (excluding comments and blank lines) and wrappers in the policy. The difference between the number of lines in F# code and resulting JavaScript shows WebSharper and its libraries at work. This allows the programmer to focus on the application logic and its security-critical parts (subject to security type check in JSLINQ) while standard boilerplate code is automatically generated by the framework. Real-world applications contain considerably more code to offer a better user experience. We omit the verification time, as execution time mostly consists of the compilation required to retrieve the AST. As the security type check is based on a simple constraint solver, we expect it to scale well to larger programs.

5. RELATED WORK

Securing web applications with IFC has been the subject of a large array of research studies. Here we contrast our approach with closely related works on IFC for web security.

Information Flow Security. Much research on formal models for end-to-end security guarantees has followed Goguen and Meseguer's seminal work on noninterference [20]. Heintze and Riecke [24] introduce the SLam calculus to enforce noninterference for a functional language with higher-order features and present a soundness proof for a functional fragment of that language. Pottier and Simonet [30] introduce a security type system for a core of ML with references and higher-order features and implement type checking for the FlowCaml tool [38]. Our framework extends the soundness proof technique from [30] with support for higher-order types, quotations and antiquotations, and declassification. A plethora of static, dynamic and hybrid analysis have been proposed to enforce noninterference-like policies [34]. Our work uses static analysis by means a security type system.

Web Application Security. Common security mechanisms proposed for web applications, including IFC, only secure components in isolation. Database systems such as MySQL provide access controls at the level of tables and columns, which are decoupled from the applications. Similarly, web browsers [23, 13] and application servers [34, 22] leverage dynamic and static techniques to enforce policies in isolation. None of these approaches can express security policies that regulate information flows across component boundaries as we do in this paper. Many existing web application frameworks augment the capabilities of a specific language with homogeneous meta-programming to ease the construction of Internet applications. WebSharper, Rails, GWT and many others are used in industry to develop complex web and mobile applications. For instance, GWT is used by many products at Google, including Flights, Hotel Finder, Offers and Wallet. While there is some framework support as prepared statements and custom sanitizers, the burden of securing code is largely placed on the developer. JSLINQ provides a smooth integration of security requirements in the development process, which allows F# programmers to check whether their code, or the code developed by external contractors, complies with desired security policies.

A few existing works aim at bridging IFC for multi-tier web applications. Chong et al. implement SIF [17] and SWIFT [16] as extensions of the JIF compiler [29] to enforce information flow policies for web applications written in Java. Web applications are checked against these policies by a combination of static and runtime enforcements. The ability to enforce fine-grained policies in the *decentralized label model* [28] is an attractive feature. At the same time, SIF and SWIFT interweave security annotations with program code and do not provide support for databases. JSLINQ addresses soundness formally and provides integration for third-party libraries. Huang et al. [25] propose WebSSARI, a tool that combines static analysis with runtime checks to detect vulnerabilities in PHP applications that interact with SQL databases. WebSSARI is very effective at discovering security vulnerabilities, although no support for client-side applications is provided and soundness is only addressed informally. Schultz and Liskov [37] propose IFDB, a database management system with decentralized IFC. IFDB is implemented by modifying PostgreSQL as well as the application environments in PHP and Python. Their *Query by Label* model provides abstractions for dealing with expressive information flow policies in relational databases, including decentralization and declassification. IFDB supports policies for server and database tiers and does not provide language integration for database queries. Corcoran et al. [18] present SELINKS which builds on the Links programming language. Links is a strongly-typed functional language for multi-tier web applications and it supports higher-order queries. SELINKS implements an expressive type system which allows to define a variety of policies, including dynamic IFC, provenance, and general access control. JSLINQ only requires the programmer write code in a mainstream language such as F# and express policies in a less sophisticated, but standard type system. Chlipala introduces UrFlow [15], which implements a static information flow analysis as part of the Ur/Web domain-specific language for development of web applications. UrFlow allows to express policies as SQL queries leveraging the users' runtime knowledge. The enforcement is done by symbolic execution over a model of the web application. UrFlow shares similar aspects with SELINKS and scalability depends on capabilities of the underlying theorem prover. While JSLINQ separates security checking from type checking, it can be extended with techniques from [43] to cope with dynamic security policies. Hedin et al. [23] present JSFlow, a security-enhanced JavaScript interpreter for fine-grained tracking of information flow. The interpreter enables deployment as a browser extension providing dynamic IFC on the client-side including third-party scripts. JSFlow only applies to applications written in JavaScript.

Secure Compilation JSLINQ relies on the WebSharper

Table 2: Comparison of web application frameworks

Tool	Client	Server	DB	3rd Party	Dec	Sound Core	Enforcement	Language	P#C
SIF/SWIFT	✓	✓	✗	✗	✓	✗	TS	Java, HTML	✗
WebSSARI	✗	✓	✓	✗	✓	✗	TS	PHP, SQL	✓
IFDB	✗	✓	✓	✗	✓	✗	Dynamic	PHP, SQL	✗
SELINKS	✓	✓	✓	✗	✓	✓	TS	Links	✗
UR/WEB	✓	✓	✓	✗	✓	✗	ATP	UR	✓
SELINQ	✗	✓	✓	✗	✗	✓	TS	F#	✗
JSFLOW	✓	✗	✗	✓	✗	✓	Dynamic	JavaScript	✗
JSLINQ	✓	✓	✓	✓	✓	✓	TS	F#	✓

compiler to translate F# code to JavaScript code deployed in the web browser, leaving out a formal investigation of the translation correctness. Fournet et al. [19] show full abstraction for a compiler which translates an ML-like language with higher-order functions and references to JavaScript. Their language is similar to F#, hence the same ideas can be used to show full abstraction for the JSLINQ compiler. Baltopoulos and Gordon [12] study secure compilation by augmenting the Links compiler with encryption and authentication for data stored on the client-side.

Tools Table 2 provides a comparison of existing web application frameworks with support for IFC. We classify each tool depending on whether they allow for IFC on the client, server, databases (DB) or third-party libraries. We also compare against support for declassification policies (Dec), soundness of a core calculus, type of enforcement mechanism (a type system (TS), a dynamic monitor or an automated theorem prover (ATP)), programming languages used and separation between code and policy (P#C). The comparison shows that JSLINQ enjoys many desirable properties.

6. CONCLUSION

We have presented a framework for end-to-end security, by leveraging IFC for a functional language with mutable store and language-integrated queries. The framework puts homogeneous meta-programming to work by developing a security type system that tracks information flows through the client, server, and underlying database. We have implemented JSLINQ and shown through different case studies that it is practical. JSLINQ can be used by organizations to build high-assurance applications. It can automatically verify the information flows within code written by internal developers or external contractors against the security policy. This helps to improve code quality and to demonstrate compliance with information security regulations, for instance when sensitive information like trade secrets or personal data is being processed. As future work, we plan to add to JSLINQ support for dynamic policies and finer-grained third-party libraries from F# and ensure their secure compilation to JavaScript.

Acknowledgments This work was funded by the European Community under the ProSecuToR project and the Swedish research agencies SSF and VR.

7. REFERENCES

[1] LINQ (Language-Integrated Query). http://msdn.microsoft.com/en-us/library/bb397926.aspx, 2014. Accessed: 2015-08-25.

[2] Apache Cordova. http://cordova.apache.org/, 2015. Accessed: 2015-09-11.

[3] Attribute-Based Mapping. https://msdn.microsoft.com/en-us/library/bb386971.aspx, 2015. Accessed: 2015-09-11.

[4] Critical Security Controls. http://www.sans.org/critical-security-controls/, 2015. Accessed: 2015-08-25.

[5] F# Compiler Services. http://fsharp.github.io/FSharp.Compiler.Service/, 2015. Accessed: 2015-09-11.

[6] FParsec. http://www.quanttec.com/fparsec/, 2015. Accessed: 2015-09-11.

[7] 'Mouse over' security flaw causes Twitter trouble. http://edition.cnn.com/2010/TECH/social.media/09/21/twitter.security.flaw/, 2015. Accessed: 2015-08-25.

[8] OWASP Top 10 2013. https://www.owasp.org/index.php/Top_10_2013-Top_10, 2015. Accessed: 2015-08-25.

[9] Sites hit in massive web attack. http://www.bbc.com/news/technology-12933053, 2015. Accessed: 2015-08-25.

[10] WebSharper. http://websharper.com/, 2015. Accessed: 2015-08-25.

[11] M. Balliu, B. Liebe, D. Schoepe, and A. Sabelfeld. JSLINQ: Building Secure Applications across Tiers. https://sites.google.com/site/jslinqcodaspy16/, September 2015. Software and Extended Version.

[12] I. G. Baltopoulos and A. D. Gordon. Secure compilation of a multi-tier web language. In *TLDI*, 2009.

[13] N. Bielova. Survey on JavaScript security policies and their enforcement mechanisms in a web browser. *JLAP*, 2013.

[14] J. Cheney, S. Lindley, and P. Wadler. A practical theory of language-integrated query. In *ICFP*, 2013.

[15] A. Chlipala. Static Checking of Dynamically-Varying Security Policies in Database-Backed Applications. In *OSDI*, 2010.

[16] S. Chong, J. Liu, A. C. Myers, X. Qi, K. Vikram, L. Zheng, and X. Zheng. Secure web applications via automatic partitioning. *Comm. of the ACM*, 2009.

[17] S. Chong, K. Vikram, and A. C. Myers. SIF: Enforcing Confidentiality and Integrity in Web Applications. In *USENIX*, 2007.

[18] B. J. Corcoran, N. Swamy, and M. W. Hicks. Cross-tier, label-based security enforcement for web applications. In *SIGMOD*, 2009.

[19] C. Fournet, N. Swamy, J. Chen, P. Dagand, P. Strub, and B. Livshits. Fully abstract compilation to javascript. In *POPL '13*, 2013.

[20] J. A. Goguen and J. Meseguer. Security Policies and Security Models. In *IEEE SP*, 1982.

[21] A. Granicz. Functional web and mobile development in F#. In *CEFP*, 2013.

[22] G. L. Guernic. *Confidentiality Enforcement Using Dynamic Information Flow Analyses.* PhD thesis, Kansas State University, 2007.

[23] D. Hedin, A. Birgisson, L. Bello, and A. Sabelfeld. JSFlow: tracking information flow in JavaScript and its APIs. In *SAC*, 2014.

[24] N. Heintze and J. G. Riecke. The SLam Calculus: Programming with Secrecy and Integrity. In *POPL*, 1998.

[25] Y.-W. Huang, F. Yu, C. Hang, C.-H. Tsai, D.-T. Lee, and S.-Y. Kuo. Securing web application code by static analysis and runtime protection. In *WWW*, 2004.

[26] X. Li and Y. Xue. A survey on server-side approaches to securing web applications. *ACM Surv.*, 2014.

[27] V. B. Livshits, A. V. Nori, S. K. Rajamani, and A. Banerjee. Merlin: specification inference for explicit information flow problems. In *PLDI*, 2009.

[28] A. C. Myers and B. Liskov. Protecting privacy using the decentralized label model. *ACM Trans. Softw. Eng. Methodol.*, 2000.

[29] A. C. Myers, L. Zheng, S. Zdancewic, S. Chong, and N. Nystrom. Jif: Java Information Flow. Software release. http://www.cs.cornell.edu/jif, July 2001.

[30] F. Pottier and V. Simonet. Information flow inference for ML. In *POPL*, 2002.

[31] S. Rasthofer, S. Arzt, and E. Bodden. A machine-learning approach for classifying and categorizing android sources and sinks. In *NDSS*, 2014.

[32] W. K. Robertson and G. Vigna. Static enforcement of web application integrity through strong typing. In *USENIX*, 2009.

[33] A. Sabelfeld and A. C. Myers. A Model for Delimited Information Release. In *ISSS*, 2003.

[34] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *JSAC*, 2003.

[35] A. Sabelfeld and D. Sands. Declassification: Dimensions and Principles. *JCS*, 2009.

[36] D. Schoepe, D. Hedin, and A. Sabelfeld. SeLINQ: tracking information across application-database boundaries. In *ICFP*, 2014.

[37] D. A. Schultz and B. Liskov. IFDB: decentralized information flow control for databases. In *EuroSys*, 2013.

[38] V. Simonet. The Flow Caml system. Software. http://cristal.inria.fr/˜simonet/soft/flowcaml, 2003.

[39] A. Stoughton, A. Johnson, S. Beller, K. Chadha, D. Chen, K. Foner, and M. Zhivich. You sank my battleship!: A case study in secure programming. 2014.

[40] D. Syme. Leveraging .NET Meta-programming Components from F#: Integrated Queries and Interoperable Heterogeneous Execution. In *ML*, 2006.

[41] D. Volpano, G. Smith, and C. Irvine. A Sound Type System for Secure Flow Analysis. *JCS*, 1996.

[42] N. Zeldovich, S. Boyd-Wickizer, and D. Mazières. Securing distributed systems with information flow control. In *5th USENIX Symposium on Networked Systems Design & Implementation, NSDI 2008, April 16-18, 2008, San Francisco, CA, USA, Proceedings*, pages 293–308, 2008.

[43] L. Zheng and A. C. Myers. Dynamic security labels and static information flow control. *Int. J. Inf. Sec.*, 2007.

APPENDIX

A. OPERATIONAL SEMANTICS

$$S ::= [\,] \mid X \mid X \mathbin{@} X$$

$$\begin{array}{rl} X ::= & \mathbf{database}(db) \mid \mathbf{yield}\ Y \mid \mathbf{if}\ Z\ \mathbf{then\ yield}\ Y \\ \mid & \mathbf{for}\ x\ \mathbf{in\ database}(db).f\ \mathbf{do}\ X \end{array}$$

$$Y ::= x \mid \{\overline{f = Z}\}$$

$$Z ::= c \mid x.f \mid op(\overline{X}) \mid \mathbf{exists}\ S$$

Figure 8: Normalized terms

$$\begin{array}{rl} v ::= & () \mid c \mid x \mid l \mid \mathbf{fun}(x) \to e \mid \mathbf{rec}\ f(x) \to e \mid (v, v) \\ \mid & \{\overline{f = v}\} \mid [\,] \mid \mathbf{yield}\ v \mathbin{@} \ldots \mathbin{@} \mathbf{yield}\ v \mid \texttt{<@}\ Q\ \texttt{@>} \end{array}$$

$$\begin{array}{rl} Q ::= & c \mid op(\overline{Q}) \mid \mathbf{lift}\ Q \mid x \mid \mathbf{fun}(x) \to Q \mid Q\ Q \mid (Q, Q) \\ \mid & \{\overline{f = Q}\} \mid Q.f \mid \mathbf{yield}\ Q \mid [\,] \mid Q \mathbin{@} Q \mid \mathbf{for}\ x\ \mathbf{in}\ Q\ \mathbf{do}\ Q \\ \mid & \mathbf{exists}\ Q \mid \mathbf{if}\ Q\ \mathbf{then}\ Q \mid \mathbf{database}(db) \end{array}$$

$$\begin{array}{rl} \mathcal{E} ::= & () \mid [\,] \mid op(\overline{v}, \mathcal{E}, \overline{e}) \mid \mathbf{lift}\ \mathcal{E} \mid \mathcal{E}\ e \mid v\ \mathcal{E} \mid (\mathcal{E}, e) \mid (v, \mathcal{E}) \\ \mid & \{\overline{f = v}, f' = \mathcal{E}, \overline{f = e}\} \mid \mathcal{E}.f \mid \mathbf{yield}\ \mathcal{E} \mid \mathcal{E} \mathbin{@} e \mid v \mathbin{@} \mathcal{E} \\ \mid & \mathbf{for}\ x\ \mathbf{in}\ \mathcal{E}\ \mathbf{do}\ e \mid \mathbf{exists}\ \mathcal{E} \mid \mathbf{if}\ \mathcal{E}\ \mathbf{then}\ e \mid \mathbf{run}\ \mathcal{E} \\ \mid & \texttt{<@}\ \mathcal{Q}[\texttt{(\%}\ \mathcal{E}\ \texttt{)}]\ \texttt{@>} \mid \mathbf{ref}\ \mathcal{E} \mid !\mathcal{E} \mid \mathcal{E} := e \mid v := \mathcal{E} \end{array}$$

$$\begin{array}{rl} \mathcal{Q} ::= & [\,] \mid op(\overline{Q}, \mathcal{Q}, \overline{e}) \mid \mathbf{fun}(x) \to \mathcal{Q} \mid \mathbf{lift}\ \mathcal{Q} \mid \mathcal{Q}\ e \mid v\ \mathcal{Q} \\ \mid & (\mathcal{Q}, e) \mid (Q, \mathcal{Q}) \mid \{\overline{f = Q}, f' = \mathcal{Q}, \overline{f = e}\} \mid \mathcal{Q}.f \\ \mid & \mathbf{yield}\ \mathcal{Q} \mid \mathcal{Q} \mathbin{@} e \mid v \mathbin{@} \mathcal{Q} \mid \mathbf{for}\ x\ \mathbf{in}\ \mathcal{Q}\ \mathbf{do}\ e \mid \mathbf{for}\ x\ \mathbf{in}\ Q\ \mathbf{do}\ \mathcal{Q} \\ \mid & \mathbf{exists}\ \mathcal{Q} \mid \mathbf{if}\ \mathcal{Q}\ \mathbf{then}\ e \mid \mathbf{if}\ Q\ \mathbf{then}\ \mathcal{Q} \mid \mathbf{run}\ \mathcal{Q} \end{array}$$

Figure 9: Values and evaluation contexts

B. SOUNDNESS PROOF

Following Pottier and Simonet [30], the noninterference proof is reduced to a subject reduction proof for an extended language and an extended type system. Noninterference requires to reason about executions of two terms e_1 and e_2, and show they are related with respect to observations at security level ℓ. The extended language provides a syntactic way to reason about execution pairs by introducing a *bracket* construct $\langle e_1 \mid e_2 \rangle$, which represents an execution pair as a single term. We refer to a term within brackets as a *binary* term and to a term without brackets and a *unary* term. Given a term e with free variables $\overline{x}$ and two related values $\overline{v_1}$ and $\overline{v_2}$, the execution of $e[\overline{v_1}/\overline{x}]$ and $e[\overline{v_2}/\overline{x}]$ can be incorporated into a term $e[\langle \overline{v_1} \mid \overline{v_2} \rangle / \overline{x}]$ in the extended language. We use this to show that two terms only differ on the confidential part if they can be encoded by a well-typed term in the extended language. Therefore, proving the noninterference of the original language is reduced to proving the subject reduction theorem of the extended language.

We extend the language syntax with the bracket construct both for terms and values. A new value **void** is used to represent cases where the memory is unbound for one of the terms, and it is compatible with any type.

$$e ::= \ldots \mid \langle e \mid e \rangle$$

$$v ::= \ldots \mid \mathbf{void} \mid \langle v \mid v \rangle$$

$$\begin{array}{rcl}
(op(\overline{v}), \mu) & \longrightarrow & \delta(op, \overline{v}, \mu) \\
((\mathbf{fun}(x) \to e)\ v, \mu) & \longrightarrow & (e[x \mapsto v], \mu) \\
((\mathbf{rec}\ f(x) \to e)\ v, \mu) & \longrightarrow & (e[f \mapsto \mathbf{rec}\ f(x) \to e, x \mapsto v], \mu) \\
(\mathbf{fst}\ (v_1, v_2), \mu) & \longrightarrow & (v_1, \mu) \\
(\mathbf{snd}\ (v_1, v_2), \mu) & \longrightarrow & (v_2, \mu) \\
(\{\overline{f = v}\}.f_i, \mu) & \longrightarrow & (v_i, \mu) \\
(\mathbf{if\ true\ then}\ v, \mu) & \longrightarrow & (v, \mu) \\
(\mathbf{if\ false\ then}\ v, \mu) & \longrightarrow & ([\,], \mu) \\
(\mathbf{if\ true\ then}\ v_1\ \mathbf{else}\ v_2, \mu) & \longrightarrow & (v_1, \mu) \\
(\mathbf{if\ false\ then}\ v_1\ \mathbf{else}\ v_2, \mu) & \longrightarrow & (v_2, \mu) \\
(\mathbf{for}\ x\ \mathbf{in\ yield}\ v\ \mathbf{do}\ e, \mu) & \longrightarrow & (e[x \mapsto v], \mu) \\
(\mathbf{for}\ x\ \mathbf{in}\ [\,]\ \mathbf{do}\ e, \mu) & \longrightarrow & ([\,], \mu) \\
(\mathbf{for}\ x\ \mathbf{in}\ e_1 \mathbin{@} e_2\ \mathbf{do}\ e_3, \mu) & \longrightarrow & ((\mathbf{for}\ x\ \mathbf{in}\ e_1\ \mathbf{do}\ e_3) \mathbin{@} \\
 & & \quad (\mathbf{for}\ x\ \mathbf{in}\ e_2\ \mathbf{do}\ e_3), \mu) \\
(\mathbf{exists}\ [\,], \mu) & \longrightarrow & (\mathbf{false}, \mu) \\
(\mathbf{exists}\ [\overline{v}], \mu) & \longrightarrow & (\mathbf{true}, \mu), \qquad |\overline{v}| > 0 \\
(\mathbf{run}\ Q, \mu) & \longrightarrow & (eval(norm(Q)), \mu) \\
(\mathbf{lift}\ c, \mu) & \longrightarrow & (\texttt{<@}\ c\ \texttt{@>}, \mu) \\
(\texttt{<@}\ \mathcal{Q}[\texttt{(\%}\ \texttt{<@}\ Q\ \texttt{@>}\ \texttt{)}]\ \texttt{@>}, \mu) & \longrightarrow & (\texttt{<@}\ \mathcal{Q}[Q]\ \texttt{@>}, \mu) \\
(\ \mathbf{ref}\ v, \mu) & \longrightarrow & (l, \mu[l \mapsto v]), \qquad l \notin dom(\mu) \\
(!l, \mu) & \longrightarrow & (\mu(l), \mu), \qquad l \in dom(\mu) \\
(l := v, \mu) & \longrightarrow & ((), \mu[l \mapsto v]), \qquad l \in dom(\mu)
\end{array}$$

$$\frac{(e, \mu) \longrightarrow (e', \mu')}{(\mathcal{E}[e], \mu) \longrightarrow (\mathcal{E}[e], \mu)}$$

Figure 10: Evaluation rules for host language

$$\begin{array}{c}
(\mathbf{fun}(x) \to R)\ Q \leadsto R[x \mapsto Q] \\
\{\overline{f = Q}\}.f_i \leadsto Q_i \\
\mathbf{for}\ x\ \mathbf{in\ yield}\ Q\ \mathbf{do}\ R \leadsto R[x \mapsto Q] \\
\mathbf{for}\ y\ \mathbf{in}\ (\mathbf{for}\ x\ \mathbf{in}\ P\ \mathbf{do}\ Q)\ \mathbf{do}\ R \leadsto \\
\qquad \mathbf{for}\ x\ \mathbf{in}\ P\ \mathbf{do}\ (\mathbf{for}\ y\ \mathbf{in}\ Q\ \mathbf{do}\ R) \\
\mathbf{for}\ x\ \mathbf{in}\ (\mathbf{if}\ P\ \mathbf{then}\ Q)\ \mathbf{do}\ R \leadsto \mathbf{if}\ P\ \mathbf{then}\ (\mathbf{for}\ x\ \mathbf{in}\ Q\ \mathbf{do}\ R) \\
\mathbf{for}\ x\ \mathbf{in}\ [\,]\ \mathbf{do}\ N \leadsto [\,] \\
\mathbf{for}\ x\ \mathbf{in}\ (P \mathbin{@} Q)\ \mathbf{do}\ R \leadsto \\
\qquad (\mathbf{for}\ x\ \mathbf{in}\ P\ \mathbf{do}\ R) \mathbin{@} (\mathbf{for}\ x\ \mathbf{in}\ Q\ \mathbf{do}\ R) \\
\mathbf{if\ true\ then}\ Q \leadsto Q \\
\mathbf{if\ false\ then}\ Q \leadsto [\,]
\end{array}$$

Figure 11: Symbolic reduction phase

The subterms of the bracket construct are either **void** or unary terms, and brackets can not be nested. *Projection* functions $\lfloor \bullet \rfloor_i$, with $i \in \{1, 2\}$, are used to establish the correspondence between binary terms and unary terms. Given a term e, the function $\lfloor e \rfloor_i = e_i$ if $e = \langle e_1 \mid e_2 \rangle$, otherwise it represents identity.

The presence of mutable storage requires to keep track of binary values shared between stores. Since memories may

$$
\begin{aligned}
\textbf{for } x \textbf{ in } P \textbf{ do } (Q \mathbin{@} R) &\hookrightarrow \\
&(\textbf{for } x \textbf{ in } P \textbf{ do } Q) \mathbin{@} (\textbf{for } x \textbf{ in } P \textbf{ do } R) \\
\textbf{for } x \textbf{ in } P \textbf{ do } [\,] &\hookrightarrow [\,] \\
\textbf{if } P \textbf{ then } (Q \mathbin{@} R) &\hookrightarrow (\textbf{if } P \textbf{ then } Q) \mathbin{@} (\textbf{if } P \textbf{ then } R) \\
\textbf{if } P \textbf{ then } [\,] &\hookrightarrow [\,] \\
\textbf{if } P \textbf{ then } (\textbf{if } Q \textbf{ then } R) &\hookrightarrow \textbf{if } P \mathbin{\texttt{\&\&}} Q \textbf{ then } R \\
\textbf{if } P \textbf{ then } (\textbf{for } x \textbf{ in } Q \textbf{ do } R) &\hookrightarrow \textbf{for } x \textbf{ in } Q \textbf{ do } (\textbf{if } P \textbf{ then } R)
\end{aligned}
$$

Figure 12: Ad-hoc reduction phase

have distinct domains, the bindings of the form $l \mapsto (v|\mathbf{void})$ and $l \mapsto (v|\mathbf{void})$ represent cases where location l is bound within only one of the two memories. The projection function is extended to memories as expected. Given a configuration (e, μ), then $\lfloor \mu \rfloor_i$ maps location l to $\lfloor \mu(l) \rfloor_i$ iff the latter is defined and is not **void**. Moreover, the projection $\lfloor (e, \mu) \rfloor_i$ is defined as $(\lfloor e \rfloor_i, \lfloor \mu \rfloor_i)$.

The operational semantics of binary terms can be expressed in terms of operational semantics of respective unary terms, as defined in the previous sections. An evaluation step of a bracket expression $\langle e_1 \mid e_2 \rangle$ is an evaluation step of either e_1 or e_2 which can only access the corresponding projection of the memory. A configuration has an index $i \in \{\bullet, 1, 2\}$ that indicates whether the term to be evaluated is a subterm of a binary term, and if so which branch of a bracket the term belongs to. For example, the configuration $\lfloor (e, \mu) \rfloor_1$, or simply $(e, \mu)_1$, means that e belongs to the first branch of a bracket, and it can only access the first projection of μ. Moreover $(e, \mu)_\bullet$, or simply (e, μ), denotes a unary configuration.

Operational Semantics.

The operational semantics rules of the extended language are given in Fig. 13. The semantics of unary reductions defined earlier (Fig. 10) applies to projections of binary terms, with a few twists regarding memory operations. The new reduction rules allow to manipulate bracket constructs, i.e., keep track of the information flows, and they do not have any computational effect on the respective projections. The purpose of *lifting* rules is to prevent the binary terms from blocking the execution. This is achieved by duplicating the shared subterm in a bracket and thus allowing the execution to proceed independently within each branch. The memory rules are modified to access the store in a context-dependent manner. In fact, the memory projection of index i forces reductions inside brackets to only affect the i-th projection of the store. The bracket construct is just a syntactic sugar to encode executions pairs and it does not have any computational effect, as shown by the following lemmas:

Lemma 1 (Soundness). *If $(e, \mu) \longrightarrow (e', \mu')$, then $\lfloor (e, \mu) \rfloor_i \longrightarrow \lfloor (e', \mu') \rfloor_i$, where $i \in \{1, 2\}$.*

Proof. The lemma can be shown by inspection of the evaluation rules. □

Lemma 2 (Completeness). *Suppose $\lfloor (e, \mu) \rfloor_i \longrightarrow^* (v_i, \mu'_i)$, where $i \in \{1, 2\}$. Then, there exists (v, μ') such that $(e, \mu) \longrightarrow^* (v, \mu')$.*

Proof. We show that (e, μ) does not admit an infinite evaluation sequence. First, infinite evaluations can not arise from lifting rules since these rules only move the bracket towards the term's root, which by definition is finite. Furthermore, lifting rules have no computational effects, hence both projections of a configuration are left unchanged. As a result, an infinite evaluation sequence only arises whenever one of the projections $\lfloor (e, \mu) \rfloor_i$ admits such an infinite sequence. But this would contraddict the assumption of the lemma, since the semantics is deterministic. On the other hand, configurations might get stuck and not produce a value. Again, we can show that (e, μ) gets stuck only if at least one of the projections $\lfloor (e, \mu) \rfloor_i$ gets stuck, which contraddicts the assumptions of the lemma. □

The completeness lemma shows that if both projections of a term can be reduced to a successful configuration, then so can the term itself. This means that we have provided enough lifting rules to allow reducing all meaningful binary terms.

Security Type System.

The security type system is extended with two typing rules to handle the bracket construct and the **void** values. Rule Bracket guarantees that binary terms are only typed in high security contexts. This reflects the intuition that binary terms encode branching under high conditions.

$$
\textsc{Bracket}\quad
\frac{\texttt{H}, \Gamma, M \vdash e_1 : t \qquad \texttt{H}, \Gamma, M \vdash e_2 : t \qquad \texttt{H} \sqsubseteq t}{pc, \Gamma, M \vdash \langle e_1 \mid e_2 \rangle : t}
$$

$$
\textsc{Void}\quad
\frac{}{pc, \Gamma, M \vdash \mathbf{void} : t}
$$

The following lemmas are needed to prove the subject reduction theorem.

Lemma 3 (Projection). *If $pc, \Gamma, M \vdash e : t$ then $pc, \Gamma, M \vdash \lfloor e \rfloor_i : t$ for $i \in \{1, 2\}$. Similarly, if $H, \Delta \vdash e : t$ then $H, \Delta \vdash \lfloor e \rfloor_i : t$.*

Proof. The lemma is proved by induction on derivation of the judgement. If e is not a bracket, the lemma follows trivially. Otherwise, suppose $e = \langle e_1 \mid e_2 \rangle$. By the premisses of the bracket rule $\texttt{H}, \Gamma, M \vdash \lfloor e \rfloor_i : t$ and since $pc \sqsubseteq \texttt{H}$, it follows that $pc, \Gamma, M \vdash \lfloor e \rfloor_i : t$. The proof for quoted judgements is similar. □

Lemma 4 (Store Access). *Let $i \in \{\bullet, 1, 2\}$ and suppose $pc, \Gamma, M \vdash v : t$ and $pc, \Gamma, M \vdash v' : t$. Moreover, if $i \in \{1, 2\}$ then $\texttt{H} \sqsubseteq t$. Then $pc, \Gamma, M \vdash$ **read**$_i$ $v : t$, $pc, \Gamma, M \vdash$ **new**$_i$ $v : t$ and $pc, \Gamma, M \vdash$ **update**$_i$ $v\ v' : t$.*

Proof. The rule follows by the definition of the auxiliary functions for the memory (Fig. 13), the projection lemma 3 and the typing rules for bracket and void constructs. We show that $pc, \Gamma, M \vdash \mathbf{new}_i\ v : t$ follows from $pc, \Gamma, M \vdash v : t$. By definition of $\mathbf{new}_i\ v$ we have three cases: (a) if $i = \bullet$, then $\mathbf{new}_\bullet\ v = v$, hence the lemma follows by assumption, (b) if $i = 1$, then $\mathbf{new}_1\ v = \langle v \mid \mathbf{void} \rangle$. By the typing rule Bracket, the projection lemma 3 and the rule Void the claim foollows immediately, (c) Symmetric to (b). □

Lemma 5 (Substitution). *Let $M \vdash v : t$ and $pc, \Gamma[x \mapsto t], M \vdash e : t'$. Then $pc, \Gamma, M \vdash e[x \mapsto v] : t'$.*

Lifting rules

$$\begin{aligned}
(\ \mathbf{ref}\ v, \mu)_i &\longrightarrow (l, \mu[l \mapsto \mathbf{new}_i\ v])_i,\ l \notin \mathit{dom}(\mu) \\
(!l, \mu)_i &\longrightarrow (\mathbf{read}_i\ \mu(l), \mu)_i,\ l \in \mathit{dom}(\mu) \\
(l := v, \mu)_i &\longrightarrow ((), \mu[l \mapsto \mathbf{update}_i\ \mu(l)\ v])_i,\ l \in \mathit{dom}(\mu) \\
(op(\overline{\langle v_1 \mid v_2\rangle}, \overline{v}), \mu) &\longrightarrow (op(\overline{\langle v_1 v \mid v_2 v\rangle}), \mu) \\
(\langle v_1 \mid v_2\rangle v, \mu) &\longrightarrow (\langle v_1 \lfloor v\rfloor_1 \mid v_2 \lfloor v \rfloor_2\rangle, \mu) \\
(!\langle l_1 \mid l_2\rangle, \mu) &\longrightarrow (\langle !l_1 \mid !l_2\rangle, \mu) \\
(\langle l_1 \mid l_2\rangle := v, \mu) &\longrightarrow (\langle l_1 := \lfloor v\rfloor_1 \mid l_2 := \lfloor v\rfloor_2\rangle, \mu) \\
(\mathbf{if}\ \langle v_1 \mid v_2\rangle\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2, \mu) &\longrightarrow (\langle \mathbf{if}\ v_1\ \mathbf{then}\ \lfloor e_1\rfloor_1\ \mathbf{else}\ \lfloor e_2\rfloor_1 \mid \mathbf{if}\ v_2\ \mathbf{then}\ \lfloor e_1\rfloor_2\ \mathbf{else}\ \lfloor e_2\rfloor_2\rangle, \mu) \\
(\mathbf{if}\ \langle v_1 \mid v_2\rangle\ \mathbf{then}\ e, \mu) &\longrightarrow (\langle \mathbf{if}\ v_1\ \mathbf{then}\ e \mid \mathbf{if}\ v_2\ \mathbf{then}\ e\rangle, \mu) \\
(\mathbf{lift}\ \langle v_1 \mid v_2\rangle, \mu) &\longrightarrow (\langle \mathbf{lift}\ v_1 \mid \mathbf{lift}\ v_2\rangle, \mu)
\end{aligned}$$

$$\frac{(e_1, \mu)_i \longrightarrow (e'_i, \mu')_i \quad e_j = e'_j \quad \{i, j\} = \{1, 2\}}{(\langle e_1 \mid e_2\rangle, \mu) \longrightarrow (\langle e'_1 \mid e'_2\rangle, \mu')}$$

Auxiliary functions

$$\begin{array}{ccc}
\mathbf{new}_\bullet\ v = v & \mathbf{update}_\bullet\ v\ v' = v' & \mathbf{read}_\bullet\ v = v \\
\mathbf{new}_1\ v = \langle v \mid \mathbf{void}\rangle & \mathbf{update}_1\ v\ v' = \langle v' \mid \lfloor v\rfloor_2\rangle & \mathbf{read}_1\ v = \lfloor v\rfloor_1 \\
\mathbf{new}_2\ v = \langle \mathbf{void} \mid v\rangle & \mathbf{update}_2\ v\ v' = \langle \lfloor v\rfloor_1 \mid v'\rangle & \mathbf{read}_2\ v = \lfloor v\rfloor_2
\end{array}$$

Figure 13: Evaluation rules for extended host language

Proof. The lemma is proved by induction on the derivation of the judgement $pc, \Gamma[x \mapsto t], M \vdash e : t'$. We show a few cases below.

Case VAR, $e = y$: If $y = x$, then since e is well typed, y occurs in the typing context and $y : t$ and $t = t'$. Moreover, $e[x \mapsto v] = v$ and $v : t'$. Otherwise, if $y \neq x$ then $e[x \mapsto v] = e$ and by assumption $e : t'$.

Case FUN, $e = \mathbf{fun}(y) \to e'$: By assumption, $pc, \Gamma[x \mapsto t], M \vdash \mathbf{fun}(y) \to e' : t_1 \overset{pc'}{\to} t_2$, where $t' = t_1 \overset{pc'}{\to} t_2$ and $v : t$. By the premise of rule FUN $pc', \Gamma[x \mapsto t][y \mapsto t'], M \vdash e' : t_2$. By induction hypothesis $pc', \Gamma, M \vdash e'[x \mapsto v] : t_2$, hence the lemma follows by the premise of FUN.

Case IF, $e = \mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3$: By assumption $pc, \Gamma[x \mapsto t], M \vdash \mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3 : t'$ and $v : t$. By induction hypothesis we have $pc, \Gamma, M \vdash e_i[x \mapsto v] : t'_i$ and by the premises of the rule IF, the claim follows.

Case BRACKET, $e = \langle e_1 \mid e_2\rangle$: By assumption, $pc, \Gamma[x \mapsto t], M \vdash \langle e_1 \mid e_2\rangle : t'$ and $v : t$. By induction hypothesis we have $\mathtt{H}, \Gamma, M \vdash e_i[x \mapsto \lfloor v\rfloor_i] : t'$, hence the lemma follows by the premises of the rule BRACKET and the projection lemma 3.

□

Theorem 3 (Subject Reduction)**.** *Let $pc, M \vdash e : t$, $M \vdash \mu$ and $(e, \mu)_i \longrightarrow (e', \mu')_i$ for $i \in \{\bullet, 1, 2\}$. Moreover, $pc = \mathtt{H}$ if $i \in \{1, 2\}$. Then there exists M' extending M, such that $pc, M' \vdash e' : t$ and $M' \vdash \mu'$.*

Proof. The theorem is shown by induction on the derivation of evaluation $(e, \mu)_i \longrightarrow (e', \mu')_i$. If the derivation of $pc, M \vdash e : t$ uses the rule SUB, then there is a $t' \sqsubseteq t$, such that $pc, M \vdash e : t'$ does not end with an instance of SUB. Hence, we can assume this is the case from now on without losing generality. Therefore, the derivation must end with a syntax directed rule which matche the term e.

Case OP, $e = op(\overline{v})$: We have $\Sigma(op) = \overline{t} \to t'$ and $e : t'^{\ell'}$, and $\overline{pc, M \vdash e : t^\ell}$ with $\ell' = \bigsqcup \ell_i$. We assume that all build in operators preserve the type, i.e. $\forall op, \overline{v : t} \Rightarrow \delta(op, \overline{v}) : t'$. Then by induction hypothesis , the typing rule OP and the assumption $M \vdash \mu$, we have that $pc, M' \vdash \delta(op, \overline{v}) : t'$ with $M = M'$.

Case FUN, $e = (\mathbf{fun}(x) \to e')\ v$: By rule APPLY we have $pc, M \vdash \mathbf{fun}(x) \to e' : t \overset{pc'}{\to} t'$ and $pc, M \vdash v : t$. By rule FUN we have that $pc', [x \mapsto t], M \vdash e' : t'$. We can then apply the substitution lemma 5 (modulo applications of rule SUB) and prove that $pc, M \vdash e'[x \mapsto v] : t'$.

Case REC, $e = (\mathbf{rec}\ f(x) \to e')\ v$: Similar to the previous case.

Case FST, $e = \mathbf{fst}\ (v_1, v_2)$: By rule FST we have $pc, M \vdash (v_1, v_2) : t_1 * t_2$ and by rule PAIR we have $pc, M \vdash v_1 : t_1$. Then the claim follows by induction hypothesis .

Case SND, $e = \mathbf{snd}\ (v_1, v_2)$: Symmetric to the previous case.

Case PROJECT, $e = \{\overline{f = v}\}.f_i$: Follows immediately by rules PROJECT, RECORD and the induction hypothesis .

Case IF1, $e = \mathbf{if}\ v_1\ \mathbf{then}\ v_2$: By rule IF1 $t = (t'\ \mathbf{list}^{\ell \sqcup \ell'})$, $pc, M \vdash v_1 : \mathbf{bool}^\ell$ and $pc, M \vdash v_2 : (t'\ \mathbf{list})^{\ell'}$. If $v_1 = \mathbf{true}$ then $e' = v_2$, otherwise $e' = []$. By induction hypothesis the claim follows.

□

To prove noninterference for values of arbitrary types, as defined by the equivalence relations $\sim_t$ in Figure 3, we need to define an encoding of input values of a given type t. The encoding transforms a pair of values $v_1 \sim_t v_2$ into a single term v using brackets whenever the component's security label is typed as high. We then prove that the resulting value has type t in the extended type system.

Definition 4 (Binary Encoding)**.** *Let v_1 and v_2 be two values such that $v_1 \sim_t v_2$. Then the encoding function Enc is recursively defined by the rules in Figure 14:*

$$\frac{v_1 \sim_{c^{\mathtt{H}}} v_2}{\langle v_1 \mid v_2 \rangle} \qquad \frac{v_1 \sim_{c^{\mathtt{L}}} v_2}{v_1} \qquad \frac{(v_1, v_2) \sim_{t_1 * t_2} (v_1', v_2')}{(v_1 \sim_{t_1} v_1', v_2 \sim_{t_2} v_2')}$$

$$\frac{\{\overline{f = v}\} \sim_{\{\overline{f:t}\}} \{\overline{f = w}\}}{\overline{v \sim_t w}} \qquad \frac{[\overline{v}] \sim_{(t\ \mathbf{list})^{\ell}} [\overline{w}]}{\ell = \mathtt{L} \Rightarrow \overline{v \sim_t w})}$$

$$\frac{[\overline{v}] \sim_{(t\ \mathbf{list})^{\ell}} [\overline{w}]}{\ell = \mathtt{H} \Rightarrow \langle [\overline{v}] \mid [\overline{w}] \rangle}$$

$$\frac{\mathbf{fun}(x) \to e_1 \sim_{t \xrightarrow{pc} t'} \mathbf{fun}(x) \to e_2 \qquad \mathtt{H} \sqsubseteq t'}{\langle \mathbf{fun}(x) \to e_1 \mid \mathbf{fun}(x) \to e_1 \rangle}$$

$$\frac{\mathbf{fun}(x) \to e_1 \sim_{t \xrightarrow{pc} t'} \mathbf{fun}(x) \to e_2 \qquad t' \sqsubseteq \mathtt{L}}{\mathbf{fun}(x) \to e_1}$$

$$\frac{\mathbf{rec}\ f(x) \to e_1 \sim_{t \xrightarrow{pc} t'} \mathbf{rec}\ f(x) \to e_2 \qquad \mathtt{H} \sqsubseteq t'}{\langle \mathbf{rec}\ f(x) \to e_1 \mid \mathbf{rec}\ f(x) \to e_1 \rangle}$$

$$\frac{\mathbf{rec}\ f(x) \to e_1 \sim_{t \xrightarrow{pc} t'} \mathbf{rec}\ f(x) \to e_2 \qquad t' \sqsubseteq \mathtt{L}}{\mathbf{rec}\ f(x) \to e_1}$$

$$\frac{v_1 \sim_{\mathbf{Expr}\langle t \rangle} v_2}{v_1 \sim_t v_2}$$

Figure 14: Value encoding

Lemma 6. *If $v_1 \sim_t v_2$ and $v = Enc(v_1, v_2, t)$, then $\vdash v : t$.*

Proof. Induction on v and rules in Figure 14. □

Then noninterference follows from the subject reduction theorem and the soundness and completeness of the extended language semantics. It is worth noting that the proof holds for multiple inputs $\overline{x : t}$, since they can be encoded as records.

Proof of Theorem 1. If $x : t \vdash e : t'$, $e[x \mapsto v_1] \longrightarrow^*_{\Omega_1} (v_1', \mu_1')$, $e[x \mapsto v_2] \longrightarrow^*_{\Omega_2} (v_2', \mu_2')$, $\Omega_1 \sim_\Sigma \Omega_2$ and $v_1 \sim_t v_2$, then $v_1' \sim_{t'} v_2'$. □

Proof. Let $v = Enc(v_1, v_2, t)$. By Lemma 6, $\vdash v : t$. By substitution lemma 5, $\vdash e[x \mapsto v] : t'$. Then since $\lfloor e[x \mapsto v] \rfloor_i = e[x \mapsto v_i]$ and, by hypothesis, evaluates to v_i' for $i \in \{1, 2\}$, we can use the completeness lemma 2 to show that $e[x \mapsto v] \longrightarrow^\star (v', \mu')$. By the subject reduction theorem 3 it follows that $\vdash v' : t'$. If v' is a bracket, we are done. Otherwise, $\lfloor v' \rfloor_1 = \lfloor v' \rfloor_2$. By the soundness theorem 1 and the determinism of the operational semantics, for $i \in \{1, 2\}$, we have $e[x \mapsto v_i] \longrightarrow^\star (v', \mu')_i$, hence $\lfloor v' \rfloor_1 = \lfloor v' \rfloor_2$. □

Proof of Theorem 2. If $x : t, \Sigma, \mathcal{D} \vdash e : t'$, $e[x \mapsto v_1] \longrightarrow^*_{\Omega_1} (v_1', \mu_1')$, $e[x \mapsto v_2] \longrightarrow^*_{\Omega_2} (v_2', \mu_2')$, $\Omega_1 \sim_\Sigma \Omega_2$, $d_j[x \mapsto v_1] \sim_\Sigma d_j[x \mapsto v_2]$ for $d_j \in \mathcal{D}$ and $\overline{v_1} \sim_t \overline{v_2}$, then $v_1' \sim_{t'} v_2'$. □

Proof. Similar to Theorem 1. The only difference is whenever rules Decl and DeclQ apply. In that case the claim follows by the assumptions of the theorem and subject reduction 3. □