SerialDetector: Principled and Practical Exploration of Object Injection Vulnerabilities for the Web

Mikhail Shcherbakov
KTH Royal Institute of Technology
mshc@kth.se

Musard Balliu
KTH Royal Institute of Technology
musard@kth.se

***Abstract*—The last decade has seen a proliferation of code-reuse attacks in the context of web applications. These attacks stem from Object Injection Vulnerabilities (OIV) enabling attacker-controlled data to abuse legitimate code fragments within a web application's codebase to execute a code chain (gadget) that performs malicious computations, like remote code execution, on attacker's behalf. OIVs occur when untrusted data is used to instantiate an object of attacker-controlled type with attacker-chosen properties, thus triggering the execution of code available but not necessarily used by the application. In the web application domain, OIVs may arise during the process of deserialization of client-side data, e.g., HTTP requests, when reconstructing the object graph that is subsequently processed by the backend applications on the server side.**

This paper presents the first systematic approach for detecting and exploiting OIVs in .NET applications including the framework and libraries. Our key insight is: The root cause of OIVs is the untrusted information flow from an application's public entry points (e.g., HTTP request handlers) to sensitive methods that create objects of arbitrary types (e.g., reflection APIs) to invoke methods (e.g., native/virtual methods) that trigger the execution of a gadget. Drawing on this insight, we develop and implement SerialDetector, a taint-based dataflow analysis that discovers OIV patterns in .NET assemblies automatically. We then use these patterns to match publicly available gadgets and to automatically validate the feasibility of OIV attacks. We demonstrate the effectiveness of our approach by an in-depth evaluation of a complex production software such as the Azure DevOps Server. We describe the key threat models and report on several remote code execution vulnerabilities found by SerialDetector, including three CVEs on Azure DevOps Server. We also perform an in-breadth security analysis of recent publicly available CVEs. Our results show that SerialDetector can detect OIVs effectively and efficiently. We release our tool publicly to support open science and encourage researchers and practitioners explore the topic further.

I. Introduction

The last decade has seen a proliferation of code-reuse attacks in the context of web applications [9], [13], [17], [18], [24], [28], [33]. The impact of these attacks can be devastating. The recent attack that hit the credit reporting agency Equifax exposed the personal information (credit card numbers, Social Security numbers) of 143 million US consumers. As a result, the law firms filed 23 class-action lawsuits, which would make it the largest suit in US history. The breach rooted in insecure deserialization in the Apache Struts framework within a Java web application, which led to remote code execution (RCE) on Equifax web servers. The attack exploited the XML serialization of complex data objects into textual strings to introduce malicious XML payloads into Struts servers during the deserialization process [46]. These attacks motivate the need for studying code-reuse vulnerabilities systematically.

Object Injection Vulnerabilities. In web applications, Object Injection Vulnerabilities (OIV) occur when an attacker can arbitrarily modify the properties of an object to abuse the data and control flow of the application. For example, OIVs may arise during the deserialization of data from the client side, e.g., HTTP requests, when reconstructing the object graph that is subsequently processed by the backend applications on the server side. Similarly to classical exploits such as return-oriented programming (ROP) and jump-oriented programming (JOP), which target memory corruption vulnerabilities [8], [36], [45], OIVs enable attacker-controlled data to trigger the execution of legitimate code fragments (gadgets) to perform malicious computations on attacker's behalf. The following *requirements* are needed to exploit an OIV [32]: (i) the attacker controls the type of the object to be instantiated, e.g., upon deserialization; (ii) the reconstructed object calls methods in the application's scope; (iii) there exists a big enough gadget space to find types that the attacker can chain to get an RCE. Existing works show that OIVs are present in mainstream programming languages and platforms like Java [24], [33], JavaScript [28], PHP [17], .NET [18], [32], and Android [34].

Challenges. Despite the high impact of OIV, efforts on tackling their root cause have been unsatisfactory. A witness is the fact that a decade after the discovery of these vulnerabilities a comprehensive understanding of languages features at the heart of OIVs has yet to emerge. One result is the ongoing arms race between researchers discovering new attacks and gadgets and vendors providing patches in an ad-hoc manner. To date, the best efforts in discovering and exploiting OIVs have been put forward by the practitioners' community [17], [18], [22], [32]. Except for a few recent works [13], [23], [25], [28], [31], the problem remains largely unexplored in the academic community. Most existing works address OIVs within the general context of injection vulnerabilities, thus lacking targeted techniques for detection and exploitation in web applications [6], [9], [43], [47].

A principled investigation of OIVs in real-world applications requires analyzing not only the applications, but also

Network and Distributed Systems Security (NDSS) Symposium 2021
21-24 February 2021
ISBN 1-891562-66-5
https://dx.doi.org/10.14722/ndss.2021.24550
www.ndss-symposium.org

the underlying framework and libraries that these applications build on. In fact, most of the known attacks stem from weaknesses in frameworks and libraries. This is challenging task since production scale frameworks, e.g., the .NET Framework, are complex entities with large codebases, intricate language features, and lack of source code. Existing approaches rely on static source code analysis of applications and ignore frameworks and libraries. Moreover, they focus on a whitelist of *magic* methods [13], [17], i.e., vulnerable APIs at the application level, thus missing attacks that may be present in unknown methods using the same features at the framework level. Another key challenge is the lack of automation and open source tools to investigate the feasibility of potential attacks. While state-of-the-art countermeasures against OIVs rely on blacklisting/whitelisting techniques [5], [10], [23], [25], [27], [31], [39], [40], it is essential to develop tools that check feasibility of attacks in a principled and practical manner.

Contributions. This work presents the first systematic approach for detecting and exploiting OIVs in .NET applications, including the .NET Framework and third-party libraries. Our key observation is that the root cause of OIVs is the untrusted information flow from an applications' *entry points* to *sensitive sinks* that create objects of arbitrary types to invoke *attack triggers* that initiate the execution of a gadget. Drawing on this insight, we develop and implement SerialDetector [41], a tool for detecting OIV *patterns* automatically and exploiting these patterns based on publicly-available gadgets in a semi-automated fashion. Following the line of work on static analysis at bytecode level [4], [7], [15], [21], [47], [48], SerialDetector implements an efficient and scalable inter-procedural taint-based static analysis targeting .NET's Common Intermediate Language. At the heart of our approach lies a field-sensitive and type-sensitive data flow analysis [42], [47] that we leverage to analyze the relevant object-oriented features and detect vulnerable patterns. We evaluate the feasibility of our approach on 15 deserializers reporting on the efficiency and effectiveness of SerialDetector in generating OIV patterns. We conduct an in-depth security analysis of production software such as the Azure DevOps Server and find three RCE vulnerabilities. To further evaluate SerialDetector, we perform an in-breadth security analysis of recent .NET CVEs from public databases and report on the effort to analyze and reproduce these exploits. In summary, the paper offers the following contributions:

- We identify the root cause of Object Injection Vulnerabilities and present a principled and practical approach to detect such vulnerabilities in a framework-agnostic manner.
- We present the first systematic approach for detecting and exploiting OIVs in .NET applications including the framework and libraries.
- We develop SerialDetector [41], a practical open source tool implementing a scalable taint-based dataflow analysis to discover OIV patterns, as well as leveraging publicly available gadgets to exploit OIVs in real-world software.
- We perform an thorough evaluation of OIV patterns in .NET-based deserialization libraries showing that SerialDetector can find vulnerable patterns with low burden on a security analysis. We use these patterns in an in-breadth security analysis of vulnerable applications to show that SerialDetector can help uncovering OIVs effectively and efficiently.
- We carry out an in-depth security analysis of Azure DevOps Server illuminating the different threat models. Drawing on these threat models, we show SerialDetector in action to identify and exploit highly-critical vulnerabilities leading to remote code execution on the server.

II. Technical Background

This section provides background information and illuminates the core security issues with OIVs in .NET applications. We identify the key ingredients in the lifecycle of an OIV, distinguishing between application-level OIVs (Section II-A) and infrastructure-level OIVs (Section II-B). Appendix A provides a brief overview of the .NET Framework.

A. Application-level OIVs

Applications can be vulnerable to OIVs whenever untrusted data instantiates an object of arbitrary type and subsequently influences a chain of method calls resulting in the execution of a dangerous operation. For an attack to be successful, the following ingredients are required: (1) a *public entry point* allowing the attacker to inject untrusted data; (2) a *sensitive method* creating an object of attacker-controlled type; (3) a *gadget* consisting of a chain of method calls that ultimately execute a dangerous operation; (4) a malicious *payload* triggering the execution of steps (1)-(3).

Consider a C# implementation of the classical *Command* design pattern [20] for a smart home controller (Listing 1). The controller implements the method `CommandAction` as an entry point handling HTTP POST requests. Following the design pattern, a developer creates an object of type `name` dynamically using the method `Activator.CreateInstance` of the .NET Framework. Subsequently, the code calls the virtual method `Execute` to execute the command specified in the input parameter `args`, e.g., a `Backup` command that runs a database backup. The main benefit of this design pattern is that a developer can define new commands without changing the implementation of the method `CommandAction`. This can be achieved by simply adding a new class that implements the interface `ICommand`.

```
public class SmartHomeController : Controller {
  [HttpPost]
  public ActionResult CommandAction(string name, string
      args) {
    var t = Type.GetType(name);
    var c = (ICommand) Activator.CreateInstance(t);
    c.Execute(args);
    return RedirectToAction("Index");
  }}
public class Backup : ICommand {
  public virtual void Execute(string parameters) {
    DB.Backup(parameters);
  }}
```

Listing 1: Implementation of Command pattern

Unfortunately, such flexible design comes with security issues. Consider the class `OSCommand` implementing the same interface `ICommand` to run a process based on the data from `parameters` (Listing 2). The method `Execute` splits the input parameters to extract the actual OS command and its arguments before the call to `Process.Start`.

```
public class OSCommand : ICommand {
  public virtual void Execute(string parameters) {
    var firstSpace = parameters.IndexOf(' ');
    var command = parameters.Substring(0, firstSpace);
    var args = parameters.Substring(firstSpace + 1);
    Process.Start(command, args);
  }}
```

Listing 2: Implementation of OSCommand

A developer might not even be aware of the existence of `OSCommand` in the modules loaded by the application. An attacker can use the class type `OSCommand` as a parameter to the POST request to create an `OSCommand` object and execute malicious commands in the target OS. For example, a payload in a POST request body with two parameters, `name = OSCommand` and `args = del /q *` results in remote code execution, deleting all files in the current directory.

Observe that the above-mentioned OIV fits our template: The application exposes a public entry point (`CommandAction`) to call a sensitive method creating an object of attacker-controlled type (`Activator.CreateInstance`). Subsequently, it uses the object to trigger the execution of a gadget (method `Execute` of class `OSCommand`) via a malicious payload. To detect such attacks, a comprehensive analysis should consider all implementations of the method `Execute` in classes implementing the `ICommand` interface.

B. *Infrastructure-level OIVs*

OIVs can be present at the level of the infrastructure that supports applications running on the server side. For .NET technologies, the infrastructure includes the .NET Framework and libraries (see Appendix A). A prime example of OIVs at the infrastructure layer is *insecure deserialization*. Deserialization is the process of recreating the original state of an object from a stream of bytes that was produced during a reverse process called serialization. In the web domain, serialization can be used to convert an object from the client side to a stream of bytes that can be transmitted over the network and used to recreate the same object on the server side. To achieve this, the deserializer may instantiate objects based on metadata from the serialized stream. Thus, an attacker can create an object of an arbitrary type by manipulating the metadata in the serialized stream, which may cause the deserializer to execute dangerous methods of the object.

We illustrate OIVs in insecure deserialization with a running example which we will discuss further in Section III. We consider the `YamlDotNet` library that implements serialization and deserialization of data in the YAML format. Listing 3 shows the simplified code fragment used by `YamlDotNet` to deserialize data obtained via the parameter `yaml`. The method `Deserialize` is a public entry point that may receive data from untrusted sources like HTTP request parameters, cookies, or files uploaded to a web application. The method parses the input and calls the method `DeserializeObject` with the root YAML node as input. A type cast ensures that the created object has the expected type `T`. However, the type cast is executed only after the creation of the object graph, hence the system will still create objects based on the information from YAML data with no restriction on the type.

```
public T Deserialize<T>(string yaml) {
  var rootNode = GetRootNode(yaml);
  return (T) DeserializeObject(rootNode);
}
private object DeserializeObject(YamlNode node) {
  var type = GetTypeFrom(node);
  var result = Activator.CreateInstance(type);
  foreach (var nestedNode in GetNestedNodes(node)) {
    var value = DeserializeObject(nestedNode);
    var property = GetPropertyOf(nestedNode);
    property.SetValue(result, value);
  }
  return result;
}
```

Listing 3: Implementation of YAML deserializer

The method `DeserializeObject` creates an object of the type specified by the YAML node and sets its fields' properties recursively. It uses a .NET Reflection API to create object by a type defined at runtime (via `Activator.CreateInstance`) and executes a setter method for each property (via `PropertyInfo.SetValue`). An attacker can find gadgets in the target system, i.e., the .NET Framework and third-party libraries, that allow executing malicious actions in their property setter. For example, the class `ObjectDataProvider` can be used as gadget for the `YamlDotNet` deserializer and any other deserializer that allows the execution of property setters for arbitrary classes.

```
public class ObjectDataProvider {
  public object ObjectInstance {
    set {
      this._objectInstance = value;
      this.Refresh();
    }}
  public void Refresh() {
    /*...*/
    obj = this._objectType.InvokeMember(
      this.MethodName, /*...*/,
      this._objectInstance, this._methodParameters);
  }}
```

Listing 4: Implementation of class ObjectDataProvider

Listing 4 shows a snippet of the class `ObjectDataProvider`. The property setter of the object `ObjectInstance` calls the method `Refresh` which in turn invokes the method specified in `MethodName` using the .NET Reflection API. Hence, the attacker controls the properties `ObjectDataProvider.MethodName` and `ObjectDataProvider.ObjectInstance` enabling the execution of arbitrary methods.

To run arbitrary commands during YAML deserialization process, e.g. a calculator, an attacker leverage the class `ObjectDataProvider` to create a payload as in Listing 5. Specifically, the deserializer will execute the property setter `ObjectDataProvider.ObjectInstance` and invoke the method `Process::Start` to run `calc.exe`.

```
!<!System.Windows.Data.ObjectDataProvider> {
  MethodName: Start,
  ObjectInstance:
  !<!System.Diagnostics.Process> {
    StartInfo:
    !<!System.Diagnostics.ProcessStartInfo> {
      FileName: cmd,
      Arguments: '/C calc.exe'
    }}}
```

Listing 5: YAML payload of ObjectDataProvider

The `YamlDotNet`'s OIV follows our template: The library exposes a public entry point (`Deserialize`) to call a sensitive method creating an object of attacker-controlled type (`Activator.CreateInstance`). Subsequently, it uses the object to trigger the execution of a gadget (the property setter of class `ObjectDataProvider`) via a malicious payload. To detect such vulnerabilities, a comprehensive analysis should consider all implementations of the property setter methods like `SetValue` in the codebase of the .NET Framework and libraries. Observe that the analysis should target .NET assemblies to account for OIVs in the framework and libraries.

III. Overview of the Approach

This section discusses the key insights of our approach (Section III-A) and provides a high-level overview of the architecture and workflow of SerialDetector (Section III-B).

A. Root cause of Object Injection Vulnerabilities

We now take a closer look at the vulnerability of YamlDotNet library in Section II-B. Listing 3 shows that the vulnerability occurs because of an insecure chain of method calls during the deserialization of attacker-controlled data. The chain starts from a call to the public method `Deserialize<T>(yaml)` which uses the untrusted input in variable `yaml` to create an object of arbitrary type via the method `Activator.CreateInstance` and subsequently use it to call the method `SetValue`. The latter executes the code of a property setter of the created object using a property name.

The vast majority of related works leverage publicly available knowledge about signatures of vulnerable methods, like `Activator.CreateInstance` and `SetValue`, to identify such *(magic)* methods in a target codebase [13], [18], [32], [33]. These works rely on the knowledge of vulnerable method signatures to either build or reuse malicious gadgets. We argue that such syntax-based approaches are not ideal as modern applications may hide unknown methods that achieve the same malicious effect. This leads us to the first research question: (i) What is an appropriate criteria for identifying OIVs? To help answering this question, we dive deeper into the analysis of the two vulnerable methods of our example.

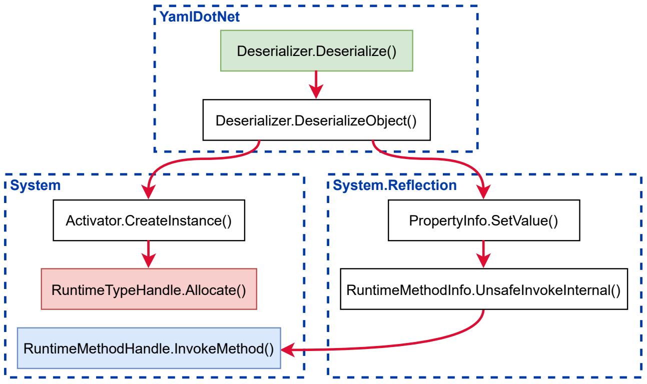

YamlDotNet
Deserializer.Deserialize()
Deserializer.DeserializeObject()
System
Activator.CreateInstance()
RuntimeTypeHandle.Allocate()
RuntimeMethodHandle.InvokeMethod()
System.Reflection
PropertyInfo.SetValue()
RuntimeMethodInfo.UnsafeInvokeInternal()

Fig. 1: OIV pattern for YamlDotNet Deserializer: public entry point (green), sensitive sink (red), and attack trigger (blue)

The method `Activator.CreateInstance` performs a sequence of method calls which results in executing the native method `RuntimeTypeHandle.Allocate(type)`. This method takes as input a parameter `type` and uses it to define the type of the returned object. We call such methods *sensitive sinks*. In general, sensitive sinks are either native (external) methods or run-time generated methods that return an object of the type specified in their input parameter. The .NET Framework contains in total 123 sensitive sinks. A similar analysis of the method `SetValue` shows that the subsequent sequence of method calls results in executing the native method `RuntimeMethodHandle.InvokeMethod(obj,..., sig)`, which invokes the method `sig` of object `obj`. Hence, an attacker controlling the type of the object `obj` and the name of the method `sig` can execute arbitrary code as in our example. We call such methods *attack triggers* since they determine the first method of a gadget chain that leads to malicious behavior. In fact, an attack trigger puts the system into a state that does not meet the specification as intended by the developer. Other potential candidates for attack triggers are virtual method calls, e.g., the method `Execute` in Listing 1, which enable attackers to execute concrete implementations of these methods at their choice.

In light of this analysis, we identify the root cause of an OIV based on three ingredients: (a) public entry points; (b) sensitive sinks; and (c) attack triggers. We use these ingredients to compute OIV *patterns* in large codebases. We define an OIV pattern as a public entry point that triggers the execution of a sensitive sink to create an object that controls the execution of an attack trigger. Figure 1 depicts the OIV pattern for our running example in Section II-B. Motivated by our notion of OIV pattern, we address three additional key questions: (ii) Can we provide practical tool support to detect OIV patterns in large-scale applications including frameworks and third-party libraries? (iii) How do we validate the usefulness of the generated patterns? (iv) Are there real-world applications to give evidence for the feasibility of the approach?

B. SerialDetector

Overview of SerialDetector. We have developed a static analysis tool, dubbed SerialDetector [41], to detect and exploit Object Injection Vulnerabilities in .NET applications and

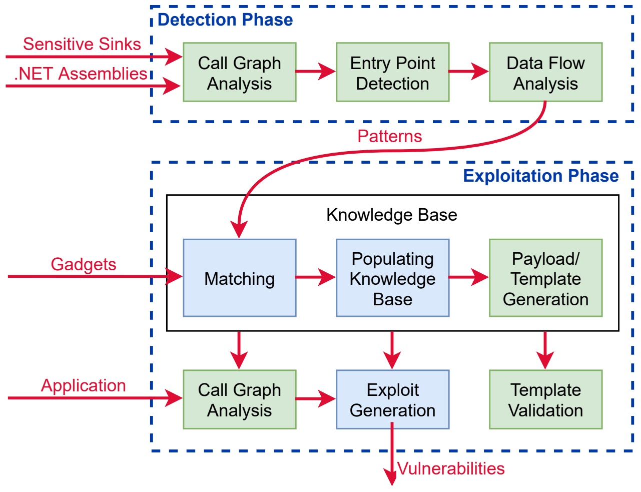


Fig. 2: Architecture and workflow of SerialDetector: automated steps (green) and manual steps (blue)

libraries. Figure 2 describes the architecture and workflow of SerialDetector. At high level, the tool operates in two phases: A fully-automated *detection* phase and a semi-automated *exploitation* phase. In the detection phase, SerialDetector takes as input a list of .NET assemblies and a list of sensitive sinks, and performs a systematic analysis to generate OIV patterns automatically. The exploitation phase matches the generated patterns with a publicly available list of gadgets. When a gadget matches a pattern, we describe the gadget in a knowledge base to generate malicious payloads for different formats. The entry points of the matched pattern allow us to describe templates in the knowledge base. Populating the knowledge base is a manual operation; the payload and template generation is performed automatically based on the described rules. For a target application, SerialDetector performs a lightweight call graph analysis to identify control flow paths that make use of the vulnerable templates described in the knowledge base. Subsequently, it uses the automatically generated payloads to validate their exploitability for the target application during the exploit generation step. The exploit generation may require modifying the payload and other application inputs, or a combination of multiple vulnerabilities into one exploit. This is a manual step requiring knowledge of the application's threat model and analysis of the data validation code, e.g., dynamic analysis or application debugging. SerialDetector does not automate this process, but provides aids such as automated validation of modified payload on a vulnerable template and automated generation of the call graph. We explain both phases in detail in Section V-A. In Section VII, we use the vulnerabilities found in the Azure DevOps Server to showcase the exploit generation and validation process.

Static analysis. SerialDetector targets the Common Intermediate Language (CIL) instead of working with the source code such as C#. This choice is motivated by several reasons: First, we aim at analyzing the code of the .NET Framework to identify sensitive methods which are not available at the source level. Second, this approach allows us to implement a framework-agnostic analysis without any knowledge about the known vulnerable methods of the framework. Third, we aim at performing an in-depth security evaluation of our approach on production software such as Microsoft Azure DevOps for which the source code is not available. Fourth, CIL has fewer language constructs that must be supported by the analyzer as compared to the high-level languages. By focusing on CIL, we do not lose any significant data that is relevant to our code analysis. In fact, CIL is a type-safe language with complete type information in the metadata. On the other hand, CIL inherits well-known challenges for the analysis of stack-based object-oriented intermediate languages, e.g., the emulation of the evaluation stack and the reconstruction of control flow.

We develop and implement a principled and practical field-sensitive taint-based dataflow analysis targeting the CIL language. In Section IV we present the details of the analysis for a core of CIL instructions. At the heart of this analysis lies a modular inter-procedural abstract interpretation based on method summaries, pointer aliasing, and efficient on-the-fly reconstruction of the control flow graph. We present the algorithms underpinning our analysis in a principled manner and discuss various challenges and solutions related to low-level language features. The analysis implements type-sensitivity, a lightweight form of context-sensitivity, and a type-hierarchy graph analysis for reconstruction of the call graph. We find that these features provide a middle ground to implementing scalable yet precise algorithms for detecting OIV patterns. Similar analysis have been implemented in the context of web applications [43], [47] and mobile applications [4], [21]. While these analysis leverage intermediate languages featuring control flow and call graph reconstruction (e.g., FlowDroid builds on the SOOT framework [48]), SerialDetector implements these features on the fly.

Roadmap of results. In Section V, we discuss our implementation of SerialDetector including challenges and limitations. Following Figure 2, the detection phase performs a call graphs analysis for a set of input assemblies, e.g., the .NET Framework and third-party libraries, to identify public entry points that may reach sensitive sinks. Then, it uses such information to carry out the dataflow analysis to identify attack triggers, thus generating a list of OIV patterns. However, the usefulness of the generated patterns depends on the existence of matching gadgets that result in exploits. While gadget generation is orthogonal to pattern generation, we evaluate SerialDetector by analyzing .NET deserialization libraries with publicly available gadgets [3]. Because an attack trigger is the first method in a gadget, it is sufficient that an attack trigger from our generated patterns matches the first method of a gadget. Subsequently, we validate the feasibility of these attacks using our payload generator. In Section VI, we discuss the details of our evaluation showing that SerialDetector finds patterns associated with vulnerable deserializers.

While these results show that SerialDetector is useful in detecting OIV patterns in the .NET Framework and its deserialization libraries, as well as in generating and validating exploits for known gadgets, it is unclear whether these vulnerabilities appear in production software. In fact, an application build on top of the .NET Framework and libraries might still use a vulnerable deserializer in a secure manner, e.g., by performing validation of the untrusted input. To validate this claim, we use SerialDetector to carry out a comprehensive in-breadth security analysis of vulnerable .NET applications (Section VI) and an in-depth security analysis of the Azure DevOps Server (Section VII). We report on the number of false positive and false negatives of our analysis, and on the number of manual changes of exploit candidates to generate a successful payload.

In Section VII we use SerialDetector's call graph analysis to identify control flow paths from public APIs of the Azure DevOps Server to vulnerable entry points in the .NET Framework. By exploring different threat models in the application, SerialDetector found three critical security vulnerabilities leading to Remote Code Execution in Azure DevOps Server. In line with the best practices of coordinated disclosure, we reported the vulnerabilities to the affected vendors. Microsoft recognized the severity of our findings and assigned CVEs to all three exploits. We also received three bug bounties acknowledging our contributions to Microsoft's security.

IV. Taint-Based Static Analysis

This section presents a taint-based static analysis underpinning the detection phase of SerialDetector. The analysis targets CIL, an object-oriented stack-based binary instruction set, and it features a modular inter-procedural field-sensitive dataflow analysis that we leverage to detect OIV patterns for large code. We provide an overview of the core language features (Section IV-A), and discuss challenges and solutions for implementing a precise, yet scalable, intra-procedural (Section IV-B) and inter-procedural analysis (Section IV-C).

A. CIL language and notation

CIL is a stack-based language running on the CLR virtual machine (see Appendix A). We focus on a subset of instructions to describe the core ideas of our analysis.

$$\begin{aligned} Inst ::= \; & \textbf{ldvar}\ x \mid \textbf{ldfld}\ f \mid \textbf{stvar}\ x \mid \textbf{stfld}\ f \mid \textbf{newobj}\ T \mid \\ & \textbf{br}\ i \mid \textbf{brtrue}\ i \mid \textbf{call}\ i \mid \textbf{ret} \end{aligned}$$

We assume a set of variables $x, y, args, \dots \in Var$ containing root variables, i.e., formal parameters of methods, and local variables; a set of object fields $f, g, \dots \in Fld$; a set of values $v, l, \dots \in Val$ consisting of object locations $l, l_1, \dots \in Loc \subseteq Val$ and other values, e.g., booleans $true$ and $false$; a set of class types $C, T \in Types$. We write $f[x \mapsto v]$ for substitution of value v for parameter x in function f and $f(x)$ for the value of x in f. We use $f(x)\downarrow$ to represent that the partial function f is defined in x, and $f(x)\uparrow$ otherwise. We write $(b\ ?\ e_1 : e_2)$ to denote a conditional expression returning e_1 if the condition b is true, e_2 otherwise.

The memory model contains an environment $E : Var \mapsto Val$ mapping variables to values, a heap $h : Loc \times Fld \mapsto Val$ mapping object locations and fields to values, an (operand) stack s and a call stack cs. The environment and heap mappings are partial functions, hence we write $\bot$ for the undefined value. A program $P \in Prog$ consists of a list of instructions $Inst^*$ indexed by a program counter index $pc, i \in PC$. We tacitly assume there is set of class definitions including a set of fields and a set of methods, and a distinguished method to start the execution. Each method definition includes a method identifier with formal parameters and the list of instructions. We write $sig \in Sig$ for the signature of a method which consists of the method's name and its formal parameters.

The execution model consists of configurations $cfg \in Conf$ of shape $cfg = (pc, cs, E, h, s)$ containing the program counter $pc \in PC$, environment $E \in Env$, heap $h \in Heap$, call stack $cs = (pc, E, s)^*$ with $cs \in (PC \times Env \times Val^*)^*$, and stack $s \in Val^*$. We write ϵ to denote an empty stack and $t :: v$ to denote a stack with top element v and tail t. The semantics of CIL programs is defined by the transition relation $\rightarrow \in Conf \times Conf$ over configurations, using the rules in Figure 12. As expected, the reflexive and transitive closure $\rightarrow^*$ of $\rightarrow$ induces a set of program executions. Notice that the program P is fixed, hence the instruction to be executed next is identified by the program counter pc. The semantics of CIL is standard and we report it in Figure 12 in Appendix.

B. Intra-procedural dataflow analysis

We now present our intra-procedural dataflow analysis based on abstract interpretation of CIL instructions. Motivated by the root cause of OIVs, our abstraction overapproximates operations over primitive types and focuses on tracking the propagation of object locations from sensitive sinks to attack triggers. Our symbolic analysis combines aliases' computation with taint tracking [37], [38] using a store-based abstraction of the heap [26]. We present the key features of the analysis implemented in SerialDetector via examples and principled rules underpinning our algorithms.

Our abstract interpretation of CIL instructions leverages a symbolic domain of values for object locations and other primitive values. Abusing notation, we assume a set of symbolic values $Val = Loc \cup Sv$ containing symbolic locations $l \in Loc$ and other symbolic values $sv \in Sv$. The latter is used as a placeholder to abstract away operations over primitive datatypes. We use symbolic configurations of shape $\langle pc, E, h, s, \phi, \psi \rangle$ where the first four components correspond to symbolic versions of the concrete counterparts, while ϕ and ψ overapproximate symbolic stacks and control flow.

Challenges and solutions at high level. Symbolic analysis for stack-based languages like CIL requires tackling several challenges related to: (a) abstract representation of the heap; (b) unstructured control flow and symbolic representation of the stack; (c) sound approximation of control flow, e.g, loops.

We address these challenges using a store-based abstraction of the heap and an efficient on-the-fly computation of merge points for conditionals and loops via forward symbolic analysis. Our analysis is flow-insensitive, hence the abstract heap graph and information about aliases holds at any program point within a method. While some code may be traversed twice to account for jump instructions, we ensure that the code is only analyzed once. Moreover, we ensure the consistency of the symbolic stack by recording the stack state for every branch instruction and combining the stacks at merge points, while updating the pointers in the heap and environment.

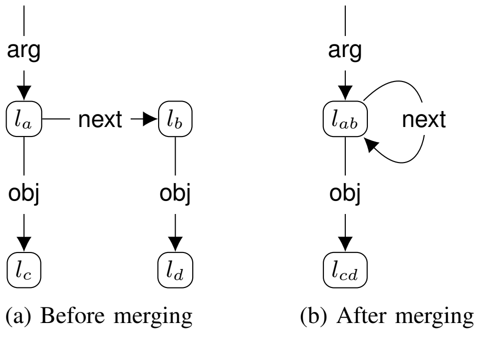
Abstracting the heap. We represent the heap as a directed graph where nodes denote abstract locations in the memory and edges describe *points-to* relations between symbolic locations. Edges contain labels corresponding to the fields and variables connecting the two locations. Here, the graph is computed from the symbolic environment and the symbolic heap.

Figure 3 depicts the abstract semantics of the heap. For simplicity, we assume that the environment E and the heap h

S-StVar
$$\frac{P(pc) = \textbf{stvar}\ x \qquad (E', h', s', \phi') = update(sv, E(x), E, h, s, \phi)}{\langle pc, E, h, s :: sv, \phi, \psi\rangle \rightarrow \langle pc+1, E', h', s', \phi', \psi\rangle}$$

S-StFld
$$\frac{P(pc) = \textbf{stfld}\ f \qquad (E', h', s', \phi') = update(h(l, f), sv, E, h, s, \phi)}{\langle pc, E, h, s :: sv :: l, \phi, \psi\rangle \rightarrow \langle pc+1, E', h', s', \phi', \psi\rangle}$$

Fig. 3: Abstract interpretation of heap



(a) Before merging (b) After merging

Fig. 4: Graph representation of symbolic heap

are initialized to fresh symbolic values $sv \in Sv$, hence $E(x)$ and $h(l, f)$ are always defined. Rules S-LdVar, S-LdFld, and S-NewObj (not shown) are similar to the corresponding rules in Figure 12 but operate on symbolic values and ignore the call stack cs. Rules S-StVar and S-StFld rely on an update function to implement the flow-insensitive and field-sensitive abstract semantics. This function takes as input two locations (as well as the current environment, heap, stack, and ϕ nodes) and merges the subgraphs rooted at those locations. The algorithm visits the subgraphs in lockstep in a breadth-first search (BFS) fashion and joins every location (node) with the same field/variable label. This is achieved by creating a fresh location and updating references to the new location. If the two merged locations have fields/variables with the same name, it recursively applies the $update$ function. Observe that the update modifies the state of the symbolic computation and may affect different components of the configuration. This approach is flow-insensitive as it updates symbolic configurations with new symbolic values, instead of overwriting the old values of the variables/fields.

```
1: arg.obj = new ClassB();
2: arg.next = new ClassA();          4a: ldvar arg  //S-LdVar
3: arg.next.obj = new ClassB();      4b: ldfld next //S-LdFld
4: arg = arg.next;                   4c: stvar arg  //S-StVar
```

Listing 6: Merging heap locations

The code snippet in Listing 6 illustrates our symbolic analysis of the heap. Our abstract interpretation yields the heap graph in Figure 4a after analyzing the (CIL representation of) instructions (1-3) in Listing 6. We now illustrate our analysis for instruction (4) and its CIL representation ($4a$-$4c$). We first load the symbolic locations in variable arg and field $next$ onto the symbolic stack by applying rules S-LdVar and S-LdFld, respectively. This results in loading the location l_b in Figure 4a. Next, we apply rule S-StVar for instruction ($4c$). The rule considers the subgraphs rooted at location l_b (the top element of the stack) and at the location l_a (since $E(arg) = l_a$) and applies the update function. Since both edges originating from the locations l_a and l_b are labeled with the field obj (which contain the locations l_c and l_d), the algorithm merges these locations to a fresh location l_{cd} and updates the graph as shown in Figure 4b.

Abstracting the control flow. The main challenge to analyzing control flow instructions is the lack of structure and the preservation of symbolic stack's consistency across different branches. We implement an analyses that does not require reconstructing of the CFG explicitly. Specifically, we analyze instructions "sequentially" following the program order imposed by the program counter pc and ensure consistency of the symbolic stack and the heap on-the-fly. To achieve this, we extend our symbolic configurations with two additional data structures: a function $\phi : PC \mapsto \wp(Stack)$ mapping program counter indexes to sets of stacks to record the symbolic stacks at the merge points of control flow branches, and a set of program counter indexes $\psi \subseteq \wp(PC)$ to record backward jumps associated with loops. The former is similar to the standard ϕ-node is high-level languages and we use it to merge the stacks corresponding to different branches in the CFG. We assume that all stacks at a merge point have the equal size, which is ensured by the high-level language compiler (e.g., the C# compiler) that translates source code to CIL code. The set ψ ensures that loops are not analyzed repeatedly. Since our analysis is flow- and path-insensitive, it suffices to analyze each basic block only once. Figure 5 illustrates our algorithm for control flow instructions. We use a function $mergeStacks : \wp(Stack) \times Heap \times Env \times \Phi \mapsto Stack \times Heap \times Env \times \Phi$ to merge all stacks and update the new symbolic configuration. Specifically, $mergeStacks$ merges symbolic locations pointwise, and updates the pointers to the merged locations in the other components.

We describe the few interesting rules in Figure 5 via examples. Consider the CIL representation of the C# ternary operator in Listing 7, which assigns the location in $var1$ or $var2$ to $arg.obj$ depending on the truth value of $flag$. The analysis should compute that field $arg.obj$ points to the merged location of variables $var1$ and $var2$. Observe that such case is not handled by the $update$ function in Figure 3. Our analysis merges the locations in $var1$ and $var2$ on the stack using rule S-StUpd. This rule has higher precedence over any other rule. Initially, $\phi(pc) = \emptyset$ for all program points. For every forward jump, as in rules S-BrFwd and S-BrTrueFwd, we store the current stack for the target instruction. For instance, the instruction at index (5), i.e., **br** 7, stores the symbolic stack containing the locations in arg and $var2$ for $\phi(7)$. When analyzing the instruction **stfld** obj at index (7), the analyzer first applies rule S-StUpd to merge the stack stored in $\phi(7)$ and the current stack, which contains the locations in arg and $var1$. Then, rule S-StFld ensures that the field $arg.obj$ contains the merged location.

$$\textsc{S-StUpd}\quad \frac{\phi(pc)\downarrow \qquad (E', h', s', \phi') = mergeStacks(\phi(pc) \cup \{s\}, E, h, \phi)}{\langle pc, E, h, s, \phi, \psi\rangle \to \langle pc, E', h', s', \phi'[pc \mapsto \bot], \psi\rangle}$$

$$\textsc{S-StSkip}\quad \frac{s = \bot}{\langle pc, E, h, s, \phi, \psi\rangle \to \langle pc+1, E, h, s, \phi, \psi\rangle}$$

$$\textsc{S-BrFwd}\quad \frac{P(pc) = \textbf{br}\ i \qquad i > pc \qquad \phi' = \phi[i \mapsto \phi(i) \cup \{s\}]}{\langle pc, E, h, s, \phi, \psi\rangle \to \langle pc+1, E, h, \bot, \phi', \psi\rangle}$$

$$\textsc{S-BrTrueFwd}\quad \frac{P(pc) = \textbf{brtrue}\ i \qquad i > pc \qquad \phi' = \phi[i \mapsto \phi(i) \cup \{s\}]}{\langle pc, E, h, s :: sv, \phi, \psi\rangle \to \langle pc+1, E, h, s, \phi', \psi\rangle}$$

$$\textsc{S-BrBwd}\quad \frac{P(pc) = \textbf{br}\ i \qquad i < pc \qquad \phi' = (pc \in \psi\ ?\ \phi : \phi[pc \mapsto s]) \qquad (pc', s', \psi') = (pc \in \psi\ ?\ (pc+1, \bot, \psi) : (i, s, \psi \cup \{pc\}))}{\langle pc, E, h, s, \phi, \psi\rangle \to \langle pc', E, h, s', \phi, \psi'\rangle}$$

$$\textsc{S-BrTrueBwd}\quad \frac{P(pc) = \textbf{brtrue}\ i \qquad i < pc \qquad \phi' = (pc \in \psi\ ?\ \phi : \phi[pc \mapsto s]) \qquad (pc', \psi') = (pc \in \psi\ ?\ (pc+1, \psi) : (i, \psi \cup \{pc\}))}{\langle pc, E, h, s :: sv, \phi, \psi\rangle \to \langle pc', E, h, s, \phi, \psi'\rangle}$$

Fig. 5: Abstract interpretation of control flow

```
// arg.obj = flag ? var1 : var2;
1: ldvar arg                // S-LdVar
2: ldvar flag               // S-LdVar
3: brtrue 6                 // S-BrTrueFwd
4: ldvar var2               // S-Ldvar
5: br 7                     // S-BrFwd
6: ldvar var1               // S-StUpd and S-LdVar
7: stfld obj                // S-StUpd and S-StFld
```

Listing 7: Ternary operator in CIL

While the previous rules ensure the consistency of the stack, we should also cater for potential loops resulting from backward jump instructions. Thanks to our flow-insensitive analysis, it suffices to analyze the "loop body" only once. Specifically, we use a set ψ to keep track of the control flow instructions that trigger a backward jump and ensure that the instructions at the jump target is analyzed only once (see S-BrBwd and S-BrTrueBwd). In particular, whenever an unconditional jump has already been analyzed, i.e. $pc \in \psi$, we set the stack to $\bot$ (undefined) and move on to executing the next instruction. An undefined stack will simply skip the analyzes of the current instruction as in rule S-StSkip unless there was another jump to that instruction with a defined stack (in which case rule S-StUpd applies)[1].

We illustrate our analysis of backward jumps with the example in Listing 8. The example models the CIL representation of the $C\#$ pattern `while(flag) {loop body}`. The analyzer examines the instruction **br** 15 at index (1) via rule S-BrFwd, recording the current stack for the instruction at index (15) in ϕ and updating the stack to undefined. This is because at this point we do not know if the next instruction at index (2) will be reached from another configuration. Therefore, we simply skip the following instructions (rule S-StSkip) until we reach a merge point, i.e., an instruction where $\phi(pc)$ is defined. In our example, the merge point is the instruction at index (15). The analyzer merges the stack in $\phi(15)$ with the undefined stack using rule S-StUpd, and uses the new stack, while updating the ϕ node. Subsequently, the analyzer loads the variable $flag$ onto the stack and examines the instruction **brtrue** 2 at index (16) via rule S-BrTrueBwd. Since $16 \notin \psi$, this results in transferring control to the instruction at index (2) and analyzing the loop body. If the analyzer reaches the instruction **brtrue** 2 again, it finds that the instruction has already been analyzed, i.e., $16 \in \psi$, and continues with the next instruction.

[1]We assume that $\phi(pc) \cup \bot = \phi(pc)$

```
 1: br 15                  // S-BrFwd
 2:
     //loop body
15:                        // S-StUpd
    // while (flag)
    ldvar flag             // S-LdVar
16: brtrue 2               // 2 x S-BrTrueBwd
```

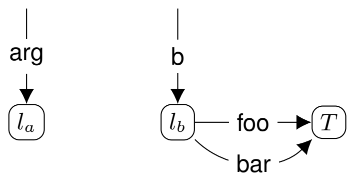
Listing 8: While loops in CIL

Aliasing and taint tracking Recall that the goal of our analyses is tracking information flows from sensitive sinks to attack triggers. To achieve this, we enrich the location nodes in our abstract heap graph with a *taint mark* whenever the return value of a sensitive sink is analyzed. Thanks to our store-based abstract heap model, the heap graph already accounts for aliases to a given node. In fact, aliases can be computed by looking at the labels of incoming edges to a given location node. Therefore, we can compute the taint mark of a reference by reading the taint mark of the node it points to.

Figure 6 provides an example of aliasing and taint tracking. The call to the sensitive sink at line (1) pushes the return value onto the stack, marks the corresponding node as tainted and adds an edge labeled with $b.foo$ to the tainted node. Similarly, the instruction at line (2) creates an alias of $b.bar$ to the tainted node, which yields the heap graph in Figure 6b. Finally, the analysis of the virtual call at line (3) reveals that the caller $b.bar$ is tainted, hence an attacker controlling its type determines which concrete implementation of $VirtualCall()$ is executed. Therefore, we consider such method as a potential attack trigger.

```
b.foo = SSink(arg);
b.bar = b.foo;
b.bar.VirtualCall();
```

(a) Code



(b) Heap

Fig. 6: Aliasing and taint tracking

C. Modular inter-procedural analysis

We now present the inter-procedural symbolic analysis underpinning our computation of OIV patterns. The analysis relies on a preliminary stage that reconstructs the Call Graph containing the entry points that may reach sensitive sinks. Subsequently, it performs a modular analysis of the call graph, based on method summaries, to determine OIV patterns.

Call graph analysis. We first analyze the target set of CIL assemblies to identify method signatures associated with **call** and **callvirt** instructions, and store them as keys in a hash table with the caller methods as values. The hash table represents a call graph, which we can reconstruct via backward analysis. A path from a sensitive sink to an entry point can be computed in $O(n)$ time, where n is the call stack's depth. We also compute the type-hierarchy graph to determine all implementations of virtual method calls. We assume that a virtual call of a base method can transfer control to any implementation of that method and store such information in the call graph. The analyzer uses this information during the backward reconstruction of the call graph from a sensitive sink to entry points, as well as during the abstract interpretation of **callvirt** instructions.

Inter-procedural analysis with method summaries. We perform a modular dataflow analysis for every entry point identified in the preliminary stage. Whenever our algorithm reaches a new method, it triggers the intra-procedural analysis (described in Section IV-B) to analyze the method independently of the caller's context, i.e., both the heap h and the environment E are empty. As a result, it produces a compact representation of the heap graph called *summary*. The summary is then stored into a caching structure K, and it is reused for every subsequent call to the same method.

We use the following notation to describe the abstract interpretation of method calls: A state $\sigma \in \mathit{State}$ is a tuple (E, h, s, ϕ, ψ) representing the calling context in a symbolic configuration and it is stored whenever we start the analysis of a new method. The symbolic call stack $cs \in (\mathit{State} \times PC)^*$ is a stack of pairs (σ, pc) containing the state of the caller and program counter index of the caller in state σ. A partial mapping $K : \mathit{Sig} \mapsto \mathit{Sum}$ caches method summaries for each method signature. A method summary $sum \in \mathit{Sum}$ is defined by the tuple (E, h) consisting of the environment and the heap.

Figure 7 presents the algorithm for our summary-based inter-procedural analysis of a call graph. We handle the following cases: (a) calls to methods with summaries already present in the cache K (rule S-CallK); (b) calls to external/native method with no implementation available (rule S-CallExt); (c) calls to (non-recursive) methods with no summaries in the cache K (rule S-Call) ; and (d) updates of the cache K upon termination of the analysis of a method (rule S-End).

S-CallK
$$\frac{P(pc) = \textbf{call}\ pc_0 \qquad K(sig_{pc_0})\downarrow \qquad \sigma' = apply(K(sig_{pc_0}), \sigma)}{\langle pc, cs, \sigma, K\rangle \rightarrow \langle pc+1, cs, \sigma', K\rangle}$$

S-Call
$$\frac{P(pc) = \textbf{call}\ pc_0 \qquad K(sig_{pc_0})\uparrow}{\langle pc, cs, \sigma, K\rangle \rightarrow \langle pc_0, cs :: (\sigma, pc), \bot, K\rangle}$$

S-CallExt
$$\frac{P(pc) = \textbf{call}\ pc_0 \qquad P(pc_0)\uparrow \qquad l \in Loc \text{ fresh}}{\langle pc, cs, \langle \_, \_, s, \_\rangle\_, K\rangle \rightarrow \langle pc+1, cs, \langle \_, \_, s :: l, \_, \_\rangle, K\rangle}$$

S-End
$$\frac{sum = cmptSum(\sigma) \qquad \sigma'' = apply(sum, \sigma') \qquad P(pc)\uparrow}{\langle pc, cs :: (\sigma', pc'), \sigma, K\rangle \rightarrow \langle pc'+1, cs, \sigma'', K[sig \mapsto sum]\rangle}$$

Fig. 7: Abstract interpretation of call graph

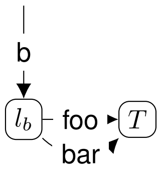
Rule S-CallK applies the cached summary of the method with signature sig_{pc_0} (at index pc_0) to the current symbolic state σ of the caller, using a function $apply : \mathit{Sum} \times \mathit{State} \mapsto \mathit{State}$. In a nutshell, $apply$ takes the root variables Var of the summary consisting of the formal parameter arg and a predefined variable $rv \in \mathit{Var}$ storing the return value of the method. Then, it pops off the top value from the stack in σ and merges it with arg using the function $update$ described in Section IV-B. The merging process may affect other components of σ that contain references to merged locations, resulting in an updated state σ'. Rule S-CallExt handles external/native method calls by pushing a fresh symbolic location onto the stack whenever a method lacks implementation, i.e., $P(pc_0)\uparrow$. Rule S-Call triggers the intra-procedural analysis of a new method by transferring control to its code at index pc_0 and storing the context of the caller in the symbolic stack cs. The caller's context contains the caller's state and program counter index pc. Observe that the analysis of the callee method is performed in a context independent manner, i.e, $\sigma' = \bot$. Rule S-Call matches rule S-End to compute the summary upon termination of the method's intra-procedural analysis (denoted by $P(pc)\uparrow$). Subsequently, it applies the summary to the caller's context σ' and caches it in K, and continues the analysis with the caller's next instruction at index $pc' + 1$.

Example: Method calls. We illustrate the abstract interpretation of non-recursive calls in Listing 8. The analysis starts from the method `EPoint` and calls `SSink` which is an external method, hence $P(pc_0)\uparrow$. Rule S-CallExt allocates a fresh location and pushes it onto the stack to emulate the return value. Because the method signature is defined as sensitive sink, we mark the fresh variable as tainted. Subsequently, the assignment stores the tainted value to the location in $b.foo$.

Next, we call the method `CreateAlias` which triggers an intra-procedural analysis of its body via rule S-Call after storing the current σ and pc to the call stack. The analysis applies rule S-StFld to create an alias between $arg.bar$ and $arg.foo$. Finally, rule S-End builds a summary from the

```
void EPoint(ClassA arg) {
  var b = new ClassB();
  b.foo = SSink(arg);
  CreateAlias(b);
  Foo(b.bar);
}
void CreateAlias(ClassB arg){
  arg.bar = arg.foo;
}
void Foo(ClassB arg) {
  ExternalMethod(arg);
}
```

(a) Code



(b) Heap

Fig. 8: Method calls

current symbolic state and stores it in the cache. The summary generation algorithm traverses the heap graph h starting from root variables Var in E and stores visited nodes and references to the summary. This is the only information that may affect the context of the caller. Subsequently, the algorithm applies the summary to the caller's state to create a new state that accounts for the effects of the method call, and proceeds with executing the next instruction of the method `EPoint`. Figure 8b depicts the effects of the summary applications, which add the edge labeled with bar to the heap graph, thus causing the two fields to point to the tainted node.

Finally, we analyze method `Foo` via rule S-CALL. `Foo` contains an external method call (as analyzed by rule S-CALLEXT) with argument arg as parameter. Since external methods can be used as attack trigger, we store information about the `ExternalMethod` in the node of the arg location. The rule S-END builds and stores the summary, and applies it to the `EPoint` context when reaching the end of the method. Hence, we merge two locations ($b.bar$ which is passed to `Foo`, and arg from the summary), and detect the call to an attack trigger with a taint mark. Finally, we store the chain from `EPoint` to `SSink` and `ExternalMethod` as an OIV pattern.

V. IMPLEMENTATION

This section provides implementation details and limitations of SerialDetector. Figure 2 overviews the architecture.

A. *Anatomy of SerialDetector*

SerialDetector [41] is written in C# and runs on the .NET platform using the dnlib library [1] for parsing assemblies.

Pattern detection. The distinguishing feature of SerialDetector is that it implements the framework-agnostic paradigm and does not use any heuristics based on method or class names to detect OIV patterns. The input consists of a set of .NET assemblies and rules for sensitive sinks and attack triggers. The sensitive sinks are initially described as a native method that return an object of type `System.Object`. Thereby, we assume that an attacker can manipulate either the parameter of the sensitive sink or the runtime state to get an object of arbitrary type. SerialDetector analyzes only CIL code in .NET assemblies and does not support binary code as in native methods. Therefore, we take a conservative approach that every native method returns an object of any derived type as the return type. We then mark the return object of the sensitive sink as tainted. The attack trigger is described as either a native (external) method that takes a tainted object as parameter or a virtual method with the first argument marked as tainted.

The pipeline of the detection phase consists of four steps: (1) SerialDetector builds an index of method call's graph for the whole .NET assembly dataset; (2) It filters all native method signatures using the criteria defining the sensitive sinks. This step yields the signatures of sensitive sinks, which we use to build the slices of the call graph in the backward direction, from the sensitive sinks to entry point methods; (3) SerialDetector performs a summary-based inter-procedural dataflow analysis as described in Section IV; (4) It outputs a sequence of patterns containing calls to attack triggers for each sensitive sink as well as traces from entry points to sensitive sinks. We collect these patterns in a knowledge base and use them as input to the exploitation phase.

Exploit generation and validation. Drawing on the knowledge base from the previous stage, we manually identify usages of vulnerable patterns in frameworks and libraries. To this end, we leverage the YSoSerial.Net project [3] to create templates that can be used to exploit vulnerabilities in a target application. We do this by declaring a signature of each public vulnerable method directly in C# code using DSL-like API. Listing 9 shows the template for the vulnerable `YamlDotNet` library from Section II-B.

```
var deserializer = new Deserializer();
Template.AssemblyVersionOlderThan(5, 0)
  .CreateBySignature(it =>
    deserializer.Deserialize(
      it.IsPayloadFrom("payload.yaml").Cast<IParser>(),
      typeof(object)));
```

Listing 9: Object Injection Template

We designed a DSL as custom LINQ expressions. LINQ is a uniform programming model for managing data in C#. Each method in the DSL call sequence refines the template model. For example, we start with the `Template` static class and call the method `AssemblyVersionOlderThan` to specify a vulnerable version of the library. The next method call `CreateBySignature` creates a template for the method `Deserialize` of the `YamlDotNet` serializer and defines as the first parameter the untrusted input with a payload from `payload.yaml`. The DSL facilitates the description of payloads and it allows to apply one payload to many templates. The key feature of the DSL is usage the *expression tree* as parameter to the method `CreateBySignature`. The expression tree represents code in an abstract syntax tree (AST), where each node is an expression. The method can extract a signature of the calling method from the expression tree, e.g., `deserializer.Deserialize`, to detect any usage in a target application. Moreover, it can also compile and run the expression tree code to test the payload. A main advantage of template generation with our DSL is that it facilitates modification and testing of different payloads, which is essential during exploitation, when SerialDetector sends a signal upon successful execution of a malicious action. SerialDetector comprises following steps for exploit generation and validation:

1. Matching (Manual): To validate the results of the detection phase, we match the generated patterns with actual sensitive sinks and attack triggers of an exploit with a known gadget. We generate a payload for the known gadgets and reproduce the exploit of each target serializer. We attach a debugger to our reproduced case and set breakpoints to the detected sensitive sink and attack trigger calls. If the breakpoints are triggered and the attack trigger performs a call chain to the malicious action of our payload, then we conclude that the pattern is exploitable.
2. Populating Knowledge Base (Manual): We use the results of the matching to populate a knowledge base. We describe the code of a gadget to create and transform to various formats to generate the payload. We also describe signatures of vulnerable entry points from the matched patterns in templates as well as additional restrictions, e.g., the version of a vulnerable library.
3. Payload and Template generation (Automated). SerialDetector automatically generates payloads and templates based on described knowledge base rules.
4. Call Graph Analysis (Automated). We use the templates as input for Call Graph Analysis to detect potentially vulnerable templates in a target application. SerialDetector generates the Call Graph from the application entry points to the vulnerable calls described in the templates.
5. Template validation (Automated). SerialDetector automatically generates and run tests for templates. It validates that a given payload can exploit an entry point in the templates. It also validates Call Graph Analysis step using template description as a source for compiling the .NET assembly with vulnerable code and it runs the analysis against this sample. All information required for testing is extracted from the knowledge base.
6. Exploit Generation (Manual). SerialDetector relies on the human-in-the-loop model for exploit generation. It provides an automatically generated call graph targeting a vulnerable template and an input payload that exploits the template. A security analyst explores the entry points of the call graph subject to attacker-controlled data, and exploits them using the original payload. The analyst may need to combine OIVs with other vulnerabilities (e.g., XSS - see Section VII-C) to execute a malicious payload for a target entry point. If an exploit fails, the analyst investigates the root cause using other tools (e.g., a debugger) and modifies the payload according to application-specific requirements.

B. Challenges and Limitations

Virtual method calls. Static analysis for large code is very challenging. We find that modularity and flow insensitivity are essential for analyzing millions of LOC. One of the challenges we faced was the analysis of virtual method calls. When performing a call graph analysis, we assume that a virtual method call may transfer control to a method of any instantiated type that implements this virtual method. For a modular data flow analysis, this means that we must analyze all implementations of the method and apply all generated summaries. To reuse merged summaries of all virtual method implementations, we introduce fake methods that include concrete calls of all implementations of a certain virtual method. We cache the summary of such method for future use.

We implement a lightweight form of context-sensitive analysis. The analyzer collects types of all created objects in a global context and then resolves the virtual method calls only for the implementations of the collected types. Because we use the modular approach we need to track summaries that have virtual calls. When a new type is instantiated, we invalidate the summaries that have the virtual calls that may be resolved to methods of the new type.

Some virtual methods of .NET Framework have hundreds of implementations. Thereby, the analysis of all implementations is a very expensive operation that often does not give us benefits. We implement several optimizations for virtual calls. Whenever possible, the analyzer infers the type of virtual calls in the intra-procedural analysis. Thereby, we can reduce the number of implementations for data flow analysis. Otherwise, we limit a count of implementations of virtual methods calls for data flow analysis and track all cases where the analyzer skips the implementations. We then perform a manual analysis of such cases and pick the ones of interest for the next run of the analysis.

Recursion. Another challenge is the modular analysis of recursion calls. The analysis must ignore caching summaries of intermediate methods in a chain of recursive methods. The reason for this is that the summaries of intermediate methods do not contain full data-flow information until we complete the analysis of the first recursive method. However, a program may have many calls of the same intermediate method, hence we must reanalyze such method, although we get the same incomplete summary. We use temporary caches for the summaries of intermediate recursive methods to analyze such methods only once within a recursion call. We then invalidate the temporary cache when the analysis of the first recursive method is completed.

Static fields. The CLI specification allows types to declare locations that are associated with those types. Such locations correspond to *static fields* of the type, hence any method has access to the static fields and can change their value. While our abstract semantics does not address static field, SerialDetector does. We enrich the summaries with an additional root variable storing the names of types with static fields. Thus, we can access any location of the static field by using such variable and the full access path. Then, we merge such root variable as we do with other arguments of the method when applying a summary to the calling method's context.

Arrays. The CLI specification defines a special type for *arrays*, providing direct support in CIL (**newarr**, **stelem**, **ldelem**, and **ldelema**). Array instructions may perform integer arithmetics when accessing an array element by taking its array index from the evaluation stack. We do not support integer arithmetics for primitive types in the current version of the analyzer. Thereby, we overapproximate the array semantics by assuming that all elements of an array point to the single abstract location containing all possible values.

Unsupported instructions. The CLI specification supports *method pointers* and *delegates* [2]. A method pointer is a type for variables that store the address of the entry point to a method. A method can be called by using a method pointer with the **calli** instruction. Delegates are the object-oriented equivalent of function pointers. Unlike function pointers, delegates are object-oriented, type-safe, and secure. Each delegate type provides a method named $Invoke$ with appropriate parameters, and each instance of a delegate forwards calls to its $Invoke$ method to one or more static or instance methods on particular objects. SerialDetector does not track values for the delegates and the method pointers, however it issues a warning whenever such features are used.

Both CLI and the .NET Framework support *reflection*. Reflection provides the ability to examine the structure of types, create instances of types, and invoke methods on types, all based on a description of the type. The current version of the analyzer does not reconstruct the call graph based on information of method invocations via the reflection.

VI. Evaluation

This section presents our experiments to validate the efficiency and effectiveness of SerialDetector. We leverage known vulnerabilities in the .NET Framework and third-party libraries as ground truth for checking the soundness and permissiveness of the detection phase, as well as for evaluating the scalability of analysis on a large codebase. To evaluate the exploitation phase, we perform an in-breadth study of deserialization vulnerabilities on real-world applications over the past two years, and report of the effort to exploit these vulnerabilities with SerialDetector. We perform the experiments on an Intel Core i7-8850H CPU 2.60GHz, 16 GB of memory, running Windows OS and .NET Framework 4.8.04084. The analysis results and data are available in SerialDetector's repository [41].

First, SerialDetector indexes all code of the .NET Framework and detects the list of sensitive sinks. The .NET Framework consists of 269 managed assemblies with 466,218 methods and 50,399 types. SerialDetector completes this step in 12.4 seconds and detects 123 different sensitive sinks. Not all sensitive sinks create new objects dynamically based on input data, hence we filter out such sensitive sinks after manual analyisis. For example, the external method `Interlocked.CompareExchange` is considered as sensitive sink, however it only implements atomic operations like comparing two objects, hence we exclude it from our list.

Detection phase. To evaluate true positives, false positives, and false negatives of the detection phase, we run SerialDetector against known OIVs in .NET Framework and third-party libraries using insecure serializers from the YSoSerial.Net project [3]. We use the deserialization methods of insecure serializers as entry points for our data flow analysis. The analyzer generates OIV patterns for each deserializer. We then match the attack triggers with gadgets from YSoSerial.Net as an indicator of effectiveness. SerialDetector confirmed exploitable patterns for 10 deserializers. It also reported warning for 5 deserializers DataContractJsonSerializer, DataContractSerializer, FsPickler, NetDataContractSerializer, and XmlSerializer since it lacks support for *delegates* calls. If a code snippet uses a delegate to create a type, we lose information about that type, hence SerialDetector cannot resolve virtual calls of that type.

Table I presents the results of our experiments. We report the *Version* of the library or the framework containing that library, and the number of different *Methods* analyzed for each entry point. The analyzer generates a summary for each method. We need re-analyze some methods, for example, recursive methods or methods with virtual calls that must be re-analyzed after creating an instance of the type with a concrete implementation. Therefore, the number of summaries is always greater than the analyzed methods.

The column *Patterns* shows the number of unique OIV patterns for each serializer, while *Priority Patterns* shows patterns that contain the methods of known gadgets. The pattern consists of the attack triggers that are called on a unique tainted object. It is unclear whether or not the rest of attack triggers is exploitable, since this requires detection of new gadgets, which we do not address in this work. Therefore, the number of (priority) patterns minus one corresponds to the number of (gadget specific) false positives.

Exploitation phase. We carry out an in-breadth analysis of .NET applications vulnerable to OIVs using the following methodology: (1) We collected vulnerabilities from the National Vulnerability Database using the keyword ".NET" and category "CWE-502 Deserialization of Untrusted Data" as of January 1st, 2019. As a result, we obtained 55 matched records; (2) We inspected the vulnerabilities manually and found that 11 vulnerabilities were actually detected in .NET applications, of which only 5 vulnerable applications were available for download; (3) We analyzed these applications with SerialDetector as reported in the first part of Table II; (4) Since not all vulnerabilities of insecure deserialization are marked as CWE-502, we searched the Internet for additional OIVs and added them in our experiments, including the new vulnerabilities that we found in Azure DevOps Server. In total, we run SerialDetector against 7 different applications with 10 OIVs. SerialDetector detected vulnerable calls of insecure deserializers and related entry points in all applications except for the Telerik UI product, which uses the Reflection API to call an insecure configuration of JavaScriptSerializer. The current version of SerialDetector does not support *reflection* for reconstructing the call graph and ignores such calls.

Table II contains information about the number of assemblies and analyzed instructions to illustrate the size of applications. The column "Entry Points w/o Threat Model" provides information about the count of all detected entry points that reach insecure serializer calls. However, not all assembly entry points are available for attackers to execute. Some are never called by an application, while others require privileges that are inaccessible to the attacker. The exploitable entry points depend on the threat model which is specific to an application. We describe the possible threat models for a web application in Section VII-B. To provide an assessment in line with the actual operation mode of SerialDetector, we leverage the (known) vulnerable entry points and compute the number of detected entry points for a specific threat model. Thus, an attacker first identifies the parts of the target system (assemblies) that are reachable for a threat model and then runs a detailed analysis. The column "Entry Points w/ Threat Model" reports the results of SerialDetector. The total number of entry points estimates the upper bound (it also includes true positives) on the number of false positives of our analysis.

	Version	Time (sec)	Memory (Mb)	Patterns	Priority Patterns	Methods	Summaries	Method Calls	Applied Summaries	Instructions
BinaryFormatter	.NET 4.8.04084	1.5	7,208	6	6	5,263	6,342	31,600	29,094	214,784
DataContractJsonSerializer	.NET 4.8.04084	122.2	16,042	73	-	14,091	16,230	112,322	102,079	576,896
DataContractSerializer	.NET 4.8.04084	51.9	13,942	73	-	13,631	15,748	109,179	99,294	562,410
FastJSON	2.3.2	3.3	7,495	24	15	6,564	7,701	41,615	37,740	273,806
FsPickler	4.6	1.5	7,216	7	-	3,552	4,302	22,927	20,362	152,343
JavaScriptSerializer	.NET 4.8.04084	44.9	13,234	121	9	18,616	19,727	130,426	120,007	665,524
LosFormatter	.NET 4.8.04084	86.3	15,278	9	9	18,941	21,631	146,864	135,843	773,037
NetDataContractSerializer	.NET 4.8.04084	158.2	17,578	72	-	14,021	15,613	104,941	96,216	545,699
Newtonsoft.Json	12.0.3	7.6	7,776	13	10	12,560	14,373	90,385	84,208	496,888
ObjectStateFormatter	.NET 4.8.04084	2.5	7,213	9	9	6,287	8,407	47,756	43,495	314,952
SharpSerializer	3.0.1	47.9	13,180	69	2	12,819	14,340	94,317	87,830	500,922
SoapFormatter	.NET 4.8.04084	8.0	7,743	12	12	11,552	12,786	79,603	73,698	444,448
XamlReader	.NET 4.8.04084	10.4	7,754	133	23	14,627	17,209	109,160	101,921	594,230
XmlSerializer	.NET 4.8.04084	158.2	16,766	82	-	14,511	16,022	114,808	106,728	583,887
YamlDotNet	4.3.1	6.0	7,754	44	2	7,253	8,441	54,581	51,080	300,192

TABLE I: Evaluation results for the insecure serializers

	Software	Version	Serializer	Entry Points w/ Threat Model (False Positives UB)	Entry Points w/o Threat Model (False Positives UB)	Assemblies/ Instructions	Payload Changes
CVE-2020-14030	Ozeki SMS Gateway	4.17.6	BinaryFormatter	31	220	84/ 1,866,312	0
CVE-2020-10915 CVE-2020-10914	VEEAM One Agent	10.0.0.750	BinaryFormatter	29	29	10/ 199,185	1
CVE-2019-18935	Telerik UI for ASP.NET AJAX	2019.2.514	JavaScriptSerializer	-	-	-	-
CVE-2019-10068	Kentico	12.0.0	SoapFormatter	1	1	191/ 5,647,128	0
CVE-2019-19470	TinyWall	2.1.8	BinaryFormatter	4	30	4/ 39,927	0
CVE-2019-0604	Microsoft SharePoint Server 2019	16.0. 10337.12109	XmlSerializer	6,283 Microsoft.SharePoint.dll; 9 Microsoft.SharePoint.Portal.dll	49,007	55/ 8,329,428	2
CVE-2019-1306	Azure DevOps Server 2019	17.143. 28621.4	BinaryFormatter	14	20	326/ 10,742,006	2
CVE-2019-0866 CVE-2019-0872	Azure DevOps Server 2019	RC2	YamlDotNet	3	13	370/ 9,863,890	1

TABLE II: Evaluation results for the real-world applications

CVE-2019-0604 in SharePoint Server has two exploitable entry points in different assemblies [49]. SerialDetector finds that both entry points and many others reach `XmlSerializer::Deserialize` call. An outlier is Microsoft.SharePoint.dll with 6,283 detected entry points. The main cause of such high complexity is the tight coupling of code in SharePoint Server and its main assembly Microsoft.SharePoint.dll, as well as our over-approximation of virtual calls. For each vulnerable entry point, we followed the approach described in Section V to generate and validate the exploits. In our experiments, we changed the payload as reported in Table II. We further clarify the practical details of threat models and exploit changes in Section VII-A.

Performance. The analysis is quite fast for such a large project as the .NET Framework. The average time of the analysis for a single serializer is 47.4 sec. This shows the advantages of our modular inter-procedural analysis. We also experimented with a whole-program dataflow analysis algorithm which did not terminate within a limit of hours. Our flow-insensitive approach reduces the size of the heap graph. This enables SerialDetector to apply summaries and merge locations faster, thus improving the overall analysis time. Another factor improving scalability is the usage of the lightweight context-sensitive analysis. Earlier versions of SerialDetector performed the analysis of virtual calls in a conservative way, analyzing all implementations of a virtual method and applying the summaries at call site. This approach generated correct patterns for very few serializers (e.g., BinaryFormatter), but it did not terminate for many others. The implementation of the type-sensitive analysis improved performance for all tested serializers.

False Positives. We also find attack triggers that are never called for a tainted object. The root cause for these false positives is flow-insensitivity of the data flow analysis. The flow-insensitive approach allows us to control the heap size at the expense of the precision of analysis. On the other hand, our results show that the number of patterns that should be reviewed manually by a security analyst is not overwhelming.

VII. In-depth Analysis of Azure DevOps Server

We evaluate SerialDetector on production software to validate its usefulness in practical scenarios. We choose Azure DevOps Server as the main target for our investigations mainly due to its complexity and diversity of threat models. Section VII-A provides a brief summary of Azure DevOps and Section VII-B provides a thorough overview of the threat

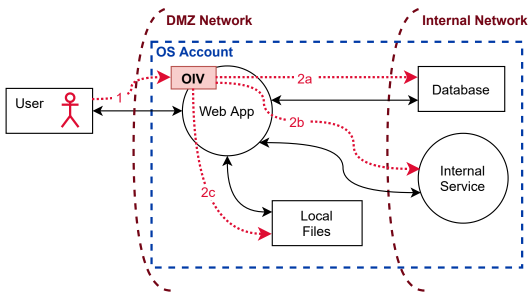


Fig. 9: First threat model

models that we explored. Section VII-C describes process of using SerialDetector to discover unknown vulnerabilities.

A. *Microsoft Azure DevOps*

The Azure DevOps Server (ADS) is a Microsoft product that provides version control, reporting, requirements management, project management, automated builds, lab management, testing, and release management capabilities. These features require integration with various data formats and serializers, thus making it an excellent target for finding OIVs. ADS hosts multiple projects across different organizations. Projects are grouped into isolated collections and the system provides functionalities to set up a project and its collections, as well as to manage users in a flexible manner. Thereby, a vulnerability that exposes high privileges in one project may lead to information disclosure of another project. ADS stores confidential information that is intellectual property (e.g., the source code of products), hence a disclosure has high impact.

ADS consists of many services exchanging information with each other, for example, the main web app, crawler and indexer services. Such system design implies complex threat models in which even internal data can be untrusted. The server has many entry points such as request handlers, documented REST APIs, plugin APIs, and internal and undocumented API. After analyzing different threat models, we use SerialDetector to automatically determine attacker-controlled entry points leading to OIVs. We then scrutinize these entry points to find RCE exploits using automated and manual analysis.

B. *Threat models*

We first consider the simple threat model of a web application running under an OS account. ADS uses the NETWORK SERVICE account in Windows by default. The code executing in the web application process has restrictions according to the OS account permissions. The web application usually has access to different services into the internal network, e.g., indexing or caching services that handle the application data. The application may also have access to a database with OS account permissions or specific credentials. Thereby, any code that executes into the web application process may have access to the database. Users communicate only with the web application in the demilitarized zone (DMZ) and do not have access to the internal network.

Figure 9 illustrates the expected information flows between services and users via black arrows. Any user can act as

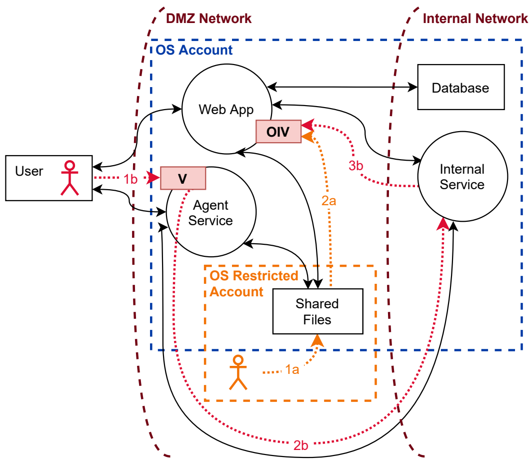


Fig. 10: Second threat model

an attacker and send payloads to the web application. If the application has an entry point that receives user data and subsequently uses code that is subject to OIVs, we can access any resources available to the OS account. This is depicted by OS Account trust boundaries in Figure 9. The attacker may send a payload to a vulnerable application directly (arrow *1*) and get access to local files (arrow *2c*), services into the internal network (arrows *2a*, *2b*) or any data from the web application memory. Example 1 illustrates this scenario.

Our second threat model addresses the question: Can an OIV be exploited if it processes data from internal services or files only? The answer depends on other components of the system. Figure 10 presents the threat model for such cases. An attacker may already be inside DMZ network and execute code with very restricted privileges. For example, the attacker's process may have access only to the shared files originating from the web application. If these files are processed by code subject to OIVs, the attacker can transfer the payload through files (arrow *1a*), escalate privileges to the web application account (arrow *2a*), and ultimately gain access to all resources inside the OS Account area in Figure 10.

Another scenario includes remote attacks through chains of vulnerabilities in other services. A service that receives untrusted user data may have vulnerabilities such as Server-Side Request Forgery (SSRF) enabling an attacker to deceive the server-side application to make requests to an arbitrary server, including internal servers. A service may also have insufficient data validation and allow to store a payload to an internal service that subsequently makes this data available to code vulnerable to OIVs. For example, an attacker may abuse a data validation vulnerability in the Agent service (arrow *1b*) and send the payload to the Internal Service (arrow *2b*). The Internal Service may index the data and send the payload to an application with OIVs (arrow *3b*). As a result, the attacker will gain access to all resources inside the OS Account area.

Our third threat model (Figure 11) targets scenarios where only a user with administrator privileges can get access to code subject to OIVs. ADS exposes web applications with a rich user management subsystem enabling the owner to create isolated projects with their own administrator accounts. We

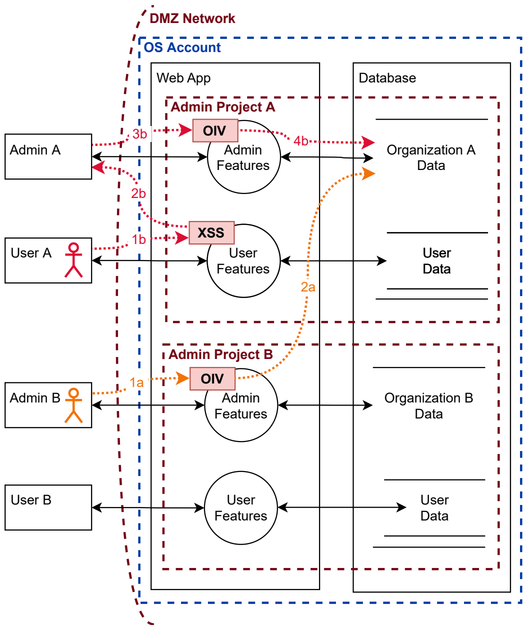


Fig. 11: Third threat model

depict this setting via the trust boundaries *Admin Project A* and *Admin Project B*. This is a typical scenario in cloud-based web applications where a user can register a separate project and become the administrator of that project. A single application process often serves several isolated projects. In this case, an attacker can register an administrator account for their own project and exploit an OIV directly (arrow *1a*) to gain access to all resources of OS Account including the database and the data of any other projects (arrow *2a*).

If the attacker has access only to a subset of features, e.g., a user with minimal privileges, they can exploit a chain of a client-side and object injection vulnerabilities to carry out the attack. For example, the attacker can exploit an XSS vulnerability to run malicious JavaScript code into the administrator's browser and use it to relay the malicious payload to OIV code that is available only to the administrator (path *1b, 2b, 3b, 4b*).

C. SerialDetector in action

We used SerialDetector to analyze the Azure DevOps Server against OIVs. We described templates of OIV in insecure serializers and run the exploitation phase of SerialDetector to determine which insecure serializers ADS uses. The tool analyzed the codebase of the application and built the Call Graph from entry points to the given insecure methods. The analyzer handled 422 assemblies that contain 630,251 methods and 11,258,350 instructions. This analysis was completed in 174 sec. Thereby, we detected an usage of 7 serializers in the codebase of ADS: BinaryFormatter, DataContractSerializer, JavaScriptSerializer, Newtonsoft.Json, XamlReader, XmlSerializer, YamlDotNet. We have checked method calls of DataContractSerializer, JavaScriptSerializer, Newtonsoft.Json, XamlReader and XmlSerializer, and concluded that it is being used in the safe mode for untrusted data.

RCE via BinaryFormatter. The BinaryFormatter matched the patterns generated by SerialDetector, hence we could instantiate objects for a malicious gadget and execute a payload. However, the BinaryFormatter handles data from local storage which an attacker cannot control directly. Following the threat model in Figure 10, SerialDetector found that one of the methods that call BinaryFormatter is located in the code of the Search Engine. The Search Engine computes indexes of text data like source files and Wiki pages to enable quick search of information. This service is a part of *Web App* in the threat model and is not accessible from outside. The indexes represent binary formatted data managed by the Storage Service. The Storage Service allows to get indexes from other components of the system and makes them available to the Search Engine. This corresponds to *Internal Service* in the threat model. A separate service Crawler tracks changes in the Git repository, parses the changed text files according to their format, and sends the resulting data to the Storage Service. The data in the Git repository is untrusted because users with minimal privileges usually have access to some repositories. This user-controlled data corresponds to *User* node in Figure 10. Hence, the security of the system relies on proper validation of the data from Git to the Crawler Service.

We analyzed the validation algorithms of the Crawler Service and identified the control flow path from the method that pulls updated Wiki pages from Git, parses the Markdown format of Wiki pages, and stores the parsed data in indexes. To exploit this path, we generated a payload with SerialDetector, stored the payload to the Wiki page, and waited for the Crawler to transfer the payload to indexes and for the Search Engine to deserialize the data using BinaryFormatter. However, the exploitation failed, hence we attached a debugger to the Agent Service to identify the instructions that changed the payload.

The Crawler first validates that the Wiki page is a text document. It uploads the file as a byte array and verifies that the content uses Unicode encoding by checking the first bytes. The payload for BinaryFormatter always starts with the byte `0x00` and the next 4 bytes contain an integer value of the ID of the root serialized object. The Crawler accepts the one sequence of the first bytes of the header that starts with `0x00` as Unicode format, and it is `0x0000FEFF`. Thereby, we changed the root ID of the payload to get the header to correspond to Unicode format, tested a new payload for BinaryFormatter using SerialDetector, and managed to bypass this validation.

We run the exploit using the new payload and failed again. Following our human-in-the-loop approach, we started a new manual iteration of the "investigating, fixing and testing" loop. The debugger revealed that the Crawler Service parses the Wiki page as Markdown document and stores the parsed data to the index. Because we use the binary data instead of a valid Markdown document, the parser rejected storing the document to the indexes. However, when the parser throws an exception, the Crawler Service stores the content of Wiki page to the index as is. This allows us to transfer the payload to the BinaryFormatter via the indexes. We found a bug in Markdown parser which throws an exception for certain incorrect strings. We then added the string to the original payload, created and tested the second version of the payload with SerialDetector, and run the exploit on ADS successfully. The attack propagates the payload from the attacker-controlled Git repository to the input of the BinaryFormatter using Crawler and Storage services, as depicted by the path *1b, 2b, 3b* in Figure 10.

We have reported the vulnerability to Microsoft following the coordinated disclosure principles. Microsoft assigned CVE-2019-1306 and released a patch to address the vulnerability. The fix uses a look-ahead approach [16] to control class loading, depending on the type name. The .NET Framework provides the class `SerializationBinder` that allows to use the look-ahead approach by configuring BinaryFormatter with a custom implementation of the binder. Thereby, a developer can create only safe types during deserialization and avoid instantiating unsafe types. The fixed version filters out the types via a whitelist which prevents the OIV exploitation.

RCE via YamlDotNet. ADS uses the YAML format for describing pipelines to automatically build and test the code of projects. The YAML pipeline configuration file may be stored in the source code repository of a project. ADS uploads the configuration file from the repository, deserializes it, and queues a build task to the Build Agent. The agent performs building and testing of the code from the repository following the YAML configuration file. For security reasons, the documentation recommends to run the agent in an isolated environment. Thus, code execution vulnerabilities during the build and test process are not directly exploitable in a typical configuration. However, the Web Application of ADS performs deserialization of the YAML file before running the agent. This boosts the impact of code execution in the Web Application to affect the entire system. For instance, an attacker can escalate to privileges of the OS Account running the Web Application.

We used SerialDetector to build the call graph of method calls that reach the YamlDotNet deserialization methods. By examining the entry points of the call graph, we found that the public Web API allows to run a build process using the YAML configuration file. We generated a payload using SerialDetector and ran the build process with our payload as the build configuration. Upon failure of our first attempt, we started the application debugging to identify a conditional statement causing the failure. The build configuration handler required small changes in the payload to pass it to the serializer. We just added the string `---` as the first and the last payload lines.

However, YAML-based pipelines were a new experimental feature at the moment and they were disabled by default. The feature can be enabled by the administrator locally on the machine. We also found an undocumented Web API to enable the feature remotely, but such request requires administrator privileges in ADS. This scenario corresponds to the threat model in Figure 11. One ADS instance supports few project collections with different user roles. However, the administrator of one collection may not have access to another collection. If the user with administrator privileges exploits the OIV and triggers an RCE, this user can get access to the resources and data of all collections. The path *1a, 2a* shows this attack.

We demonstrated higher impact of the YamlDotNet OIV by looking for XSS vulnerabilities. We found two XSSs using static and manual analysis. The first one can be exploited when the victim opens a PDF file from the source code repository using the ADS web interface. We use a weakness of Internet Explorer to execute scripts embedded into PDF files (now this is also fixed). Thereby, an attacker needs to prepare a malicious PDF file, upload it to the repository, and craft the link to the PDF file using the viewer of ADS. When the administrator opens this link in Internet Explorer, the embedded script sends requests with administrator privileges to ADS triggering the deserialization of the malicious YAML file. Thus, the attacker executes an RCE attack on the target ADS with minimal privileges (i.e., only access to the source code repository).

The second XSS targets a victim that opens a page with the test description. ADS uses the Test hub feature for tracking the manual testing of applications. It provides three main types of test management artifacts: test plans, test suites, and test cases. The test case description field had insufficient validation and sanitization of the input text. The attacker may inject JavaScript in the description field and get a stored XSS on the Test Case page. When the administrator opens this page, the JavaScript code is executed in the administrator's browser allowing for requests to Web API with administrator privileges. We exploited the vulnerability similarly to the RCE on the server. The path *1b, 2b, 3b, 4b* illustrates the attack.

We reported these vulnerabilities to Microsoft following the coordinated disclosure principles. Microsoft assigned CVE-2019-0866 and CVE-2019-0872 for each vulnerable attack chain and fixed it. The XSS vulnerabilities were fixed by adding additional validation to the web page and by requiring users to download the PDF document instead of opening it in the browser. To prevent the OIV exploitation, Microsoft implemented their own lightweight YAML serializer using a parser from the YamlDotNet. This serializer does not allow to instantiate an object based on the type of information from the file. It deserializes only a small predefined subset of types which prevents the composition of a malicious gadget.

VIII. Related works

This section discusses related works targeting object injection vulnerabilities and injection vulnerabilities.

Object Injection Vulnerabilities. The closest related research is the work of Dahse et al. [11], [13], which implements static analysis to systematically detect gadgets in common PHP applications. Like us, they implement static taint analysis to detect exploitable vulnerabilities. The key difference is that SerialDetector's analysis operates at the assembly level to discover new OIV patterns, while Dahse et al. target PHP source code via well-known attack triggers (called magic methods in their setting). On the other hand, SerialDetector relies on known gadgets. An interesting avenue for future work is to explore the complementary techniques by Dahse et al. to implement gadget generation in SerialDetector.

Shahriar and Haddad [40] propose a lightweight approach based on latent semantic indexing to identify keywords that are likely responsible for OIVs and apply it systematically to PHP applications to find new vulnerabilities. Rasheed et al. [35] study DoS vulnerabilities in YAML libraries across different programming languages and discover several new vulnerabilities. Recently, Lekies et al. [28] showed that code-reuse attacks are feasible in the client-side web applications by proposing a new attack vector that breaks all existing XSS mitigations via script gadgets. Cristalli et al. [10] propose a dynamic approach to identify trusted execution paths during a training phase with benign inputs, and leverages this information to detect insecure deserialization via a lightweight sandbox. Hawkins and Demsky [23] present ZenIDS, a system to dynamically learn the trusted execution paths of an application during an online training period and report execution anomalies as potential intrusions. Dietrich et al. [14] investigate deserialization vulnerabilities to exploit the topology of object graphs constructed from Java classes in a way that leads dererialization to DOS attacks exhausting stack memory, heap memory, and CPU time. SerialDetector focuses on generating OIV patterns targeting low level features of the framework and libraries. Our results are complementary and can help improve the precision of these techniques. Moreover, to our best knowledge, none of the existing static analysis has been applied to complex production software such as Azure DevOps Server.

Our work draws inspiration on exploitation techniques developed by the practitioners' community [17], [18], [22], [32]. We leverage these results for the exploitation phase to match our patterns with existing gadgets [3]. We refer to Muñoz and Mirosh [32] for an excellent report on deserialization attacks in .NET and Java libraries. Seacord [39] provides a thorough discussion on OIV defenses via type whitelisting. Our results are complementary to gadget generation techniques and can help these works uncovering unknown gadgets.

Tool support Koutroumpouchos et al. [27] develop ObjectMap, a toolchain for detecting and testing OIVs in Java and PHP applications. While targeting different languages, ObjectMap shares similar goals as SerialDetector's payload and exploit generation modules. Gadget Inspector [22] is a tool for discovering gadget chains that can be used to exploit deserialization vulnerabilities in Java applications. SerialKiller [33] is a Java deserialization library implementing look-ahead dererialization [16] to secure applications from untrusted input. It inspects Java classes during naming resolution and allows a combination of blacklisting and whitelisting.

Injection Vulnerabilities Code reuse vulnerabilities have been studied in breadth in the context of injection vulnerabilities in web applications [6], [9], [12], [24], [28]–[30], [43], [44], [47], [47]. For the .NET domain, Fu et al. [19] propose the design of a symbolic execution framework for .NET bytecode to identify SQL injection vulnerabilities. Doupé et al. [15] implement a semantics-preserving static refactoring analysis to separate code and data in .NET binaries with the goal of protecting legacy applications from server-side XSS attacks. Our work is exclusively focused on OIVs and yields results that target such vulnerability in depth. Except for significant engineering challenges with .NET assemblies (including the framework and libraries), our taint-based data flow analysis follows the existing line of work targeting web and mobile application vulnerabilities at the bytecode level broadly [4], [7], [21], [30], [43], [47].

IX. Conclusion

We have pushed the research boundary on key challenges for OIVs in the modern web. Based on these challenges, we have identified the root cause of OIV and proposed patterns based on the triplet: entry points, sensitive sinks, and attack triggers. We have presented SerialDetector, the first principled and practical tool implementing a systematic exploration of OIVs via taint-based static analysis. We have used SerialDetector to test 15 serialization libraries as well as several vulnerable applications. We have performed an in-depth security analysis of the Azure DevOps Server which led SerialDetector discover RCE vulnerabilities with three assigned CVEs.

Acknowledgment

We thank the anonymous reviewers for useful feedback. The work was partly funded by the Swedish Research Council (VR) under the project JointForce and by the Swedish Foundation for Strategic Research (SSF) under the project TrustFull.

References

[1] "dnlib," https://github.com/0xd4d/dnlib.

[2] "Standard ECMA-335 Common Language Infrastructure (CLI)," https://www.ecma-international.org/publications/standards/Ecma-335.htm.

[3] "YSoSerial.Net," https://github.com/pwntester/ysoserial.net.

[4] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "FlowDroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps," in *PLDI 2014*, 2014, p. 29.

[5] B. A. Azad, P. Laperdrix, and N. Nikiforakis, "Less is more: Quantifying the security benefits of debloating web applications," in *USENIX Security 19*, 2019, pp. 1697–1714.

[6] M. Backes, K. Rieck, M. Skoruppa, B. Stock, and F. Yamaguchi, "Efficient and flexible discovery of php application vulnerabilities," in *EuroS&P'17*, 2017, pp. 334–349.

[7] M. Balliu, D. Schoepe, and A. Sabelfeld, "We are family: Relating information-flow trackers," 2017, pp. 124–145.

[8] T. K. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang, "Jump-oriented programming: a new class of code-reuse attack," in *ASIACCS 2011*, 2011, pp. 30–40.

[9] C. Cifuentes, A. Gross, and N. Keynes, "Understanding caller-sensitive method vulnerabilities: A class of access control vulnerabilities in the java platform," in *SOAP 2015*, 2015, pp. 7–12.

[10] S. Cristalli, E. Vignati, D. Bruschi, and A. Lanzi, "Trusted Execution Path for Protecting Java Applications Against Deserialization of Untrusted Data," in *RAID 2018*, 2018, pp. 445–464.

[11] J. Dahse and T. Holz, "Simulation of built-in PHP features for precise static code analysis," in *NDSS'14*, 2014.

[12] ——, "Static detection of second-order vulnerabilities in web applications," in *USENIX Security 14*, 2014, pp. 989–1003.

[13] J. Dahse, N. Krein, and T. Holz, "Code reuse attacks in php: Automated pop chain generation," in *CCS'14*, 2014, pp. 42–53.

[14] J. Dietrich, K. Jezek, S. Rasheed, A. Tahir, and A. Potanin, "Evil Pickles: DoS Attacks Based on Object-Graph Engineering," in *ECOOP 2017*, 2017, pp. 10:1–10:32.

[15] A. Doupé, W. Cui, M. H. Jakubowski, M. Peinado, C. Kruegel, and G. Vigna, "dedacota: toward preventing server-side XSS via automatic code and data separation," in *CCS'13*, 2013, pp. 1205–1216.

[16] P. Ernst, "Look-ahead Java deserialization," January 2013. [Online]. Available: https://www.ibm.com/developerworks/library/se-lookahead/

[17] S. Esser, "Utilizing code reuse/rop in php application exploits," *Black-Hat USA*, 2010.

[18] J. Forshaw, "Are you my Type? Breaking .NET Through Serialization," BlackHat, 2012.

[19] X. Fu, X. Lu, B. Peltsverger, S. Chen, K. Qian, and L. Tao, "A static analysis framework for detecting SQL injection vulnerabilities," in *COMPSAC 2007*, 2007, pp. 87–96.

[20] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. USA: Addison-Wesley Longman Publishing Co., Inc., 1995.

[21] M. I. Gordon, D. Kim, J. H. Perkins, L. Gilham, N. Nguyen, and M. C. Rinard, "Information flow analysis of android applications in droidsafe," in *NDSS*, 2015.

[22] I. Haken, "Automated Discovery of Deserialization Gadget Chains," BlackHat, 2018.

[23] B. Hawkins and B. Demsky, "Zenids: introspective intrusion detection for PHP applications," in *ICSE 2017*, 2017, pp. 232–243.

[24] P. Holzinger, S. Triller, A. Bartel, and E. Bodden, "An in-depth study of more than ten years of java exploitation," in *CCS'16*, 2016, pp. 779–790.

[25] J. Huang, Y. Li, J. Zhang, and R. Dai, “Uchecker: Automatically detecting php-based unrestricted file upload vulnerabilities,” in *DSN 2019*, 2019, pp. 581–592.

[26] V. Kanvar and U. P. Khedker, “Heap abstractions for static analysis,” *ACM Comput. Surv.*, vol. 49, no. 2, June 2016.

[27] N. Koutroumpouchos, G. Lavdanis, E. Veroni, C. Ntantogian, and C. Xenakis, “ObjectMap: Detecting Insecure Object Deserialization,” in *PCI’19*, 2019, pp. 67–72.

[28] S. Lekies, K. Kotowicz, S. Groß, E. A. V. Nava, and M. Johns, “Code-reuse attacks for the web: Breaking cross-site scripting mitigations via script gadgets,” in *CCS 2017*, 2017, pp. 1709–1723.

[29] S. Lekies, B. Stock, and M. Johns, “25 million flows later: large-scale detection of DOM-based XSS,” in *CCS 2013*, 2013, pp. 1193–1204.

[30] W. Melicher, A. Das, M. Sharif, L. Bauer, and L. Jia, “Riding out DOMsday: Toward detecting and preventing DOM cross-site scripting,” in *NDSS 2018*, 2018.

[31] D. Mitropoulos, P. Louridas, M. Polychronakis, and A. D. Keromytis, “Defending against web application attacks: Approaches, challenges and implications,” *IEEE Transactions on Dependable and Secure Computing*, vol. 16, no. 2, pp. 188–203, 2019.

[32] A. Muñoz and O. Mirosh, “Friday the 13th JSON Attacks,” BlackHat, 2017.

[33] A. Muñoz and C. Schneider, “Serial killer: Silently pwning your java endpoints,” 2018.

[34] O. Peles and R. Hay, “One class to rule them all: 0-day deserialization vulnerabilities in android,” in *WOOT’15*, 2015.

[35] S. Rasheed, J. Dietrich, and A. Tahir, “Laughter in the wild: A study into dos vulnerabilities in YAML libraries,” in *TrustCom/BigDataSE 2019*, 2019, pp. 342–349.

[36] R. Roemer, E. Buchanan, H. Shacham, and S. Savage, “Return-oriented programming: Systems, languages, and applications,” *ACM Trans. Inf. Syst. Secur.*, vol. 15, no. 1, pp. 2:1–2:34, 2012.

[37] D. Schoepe, M. Balliu, B. C. Pierce, and A. Sabelfeld, “Explicit secrecy: A policy for taint tracking,” in *EuroS&P*, 2016.

[38] E. J. Schwartz, T. Avgerinos, and D. Brumley, “All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask),” in *IEEE S&P*, 2010.

[39] R. Seacord, “Combating Java Deserialization Vulnerabilities with Look-Ahead Object Input Streams (LAOIS),” June 2017.

[40] H. Shahriar and H. Haddad, “Object injection vulnerability discovery based on latent semantic indexing,” in *SAC*, 2016, pp. 801–807.

[41] M. Shcherbakov and M. Balliu, “SerialDetector,” February 2021, software. [Online]. Available: https://github.com/yuske/SerialDetector

[42] Y. Smaragdakis, M. Bravenboer, and O. Lhoták, “Pick your contexts well: understanding object-sensitivity,” in *POPL 2011*, 2011, pp. 17–30.

[43] F. Spoto, E. Burato, M. D. Ernst, P. Ferrara, A. Lovato, D. Macedonio, and C. Spiridon, “Static identification of injection attacks in java,” *ACM Trans. Program. Lang. Syst.*, vol. 41, no. 3, pp. 18:1–18:58, 2019.

[44] C. Staicu and M. Pradel, “Freezing the web: A study of ReDoS vulnerabilities in JavaScript-based web servers,” in *USENIX Security*, 2018, pp. 361–376.

[45] L. Szekeres, M. Payer, T. Wei, and D. Song, “SoK: Eternal War in Memory,” in *Security & Privacy*, 2013, pp. 48–62.

[46] K. TEAM, “OWASP Top 10 2017 – A8 Insecure Deserialization,” https://www.kiuwan.com/blog/owasp-top-10-2017-a8-insecure-deserialization/.

[47] O. Tripp, M. Pistoia, P. Cousot, R. Cousot, and S. Guarnieri, “Andromeda: Accurate and scalable security analysis of web applications,” in *FASE*, 2013, pp. 210–225.

[48] R. Vallée-Rai, P. Co, E. Gagnon, L. J. Hendren, P. Lam, and V. Sundaresan, “Soot - a java bytecode optimization framework,” in *CASCON*, 1999.

[49] M. Wulftange, “CVE-2019-0604: Details of a Microsoft SharePoint RCE Vulnerability,” 2019. [Online]. Available: https://www.thezdi.com/blog/2019/3/13/cve-2019-0604-details-of-a-microsoft-sharepoint-rce-vulnerability

$$\textsc{C-LdVar}\quad \frac{P(pc) = \textbf{ldvar}\ x \qquad v = E(x)}{\langle pc, cs, E, h, s\rangle \rightarrow \langle pc+1, cs, E, h, s :: v\rangle}$$

$$\textsc{C-LdFld}\quad \frac{P(pc) = \textbf{ldfld}\ f \qquad v = h(l, f)}{\langle pc, cs, E, h, s :: l\rangle \rightarrow \langle pc+1, cs, E, h, s :: v\rangle} \qquad \textsc{C-Br}\quad \frac{P(pc) = \textbf{br}\ i}{\langle pc, cs, E, h, s\rangle \rightarrow \langle i, cs, E, h, s\rangle}$$

$$\textsc{C-StVar}\quad \frac{P(pc) = \textbf{stvar}\ x \qquad E' = E[x \mapsto v]}{\langle pc, cs, E, h, s :: v\rangle \rightarrow \langle pc+1, cs, E', h, s\rangle}$$

$$\textsc{C-StFld}\quad \frac{P(pc) = \textbf{stfld}\ f \qquad h' = h[h(l, f) \mapsto v]}{\langle pc, cs, E, h, s :: v :: l\rangle \rightarrow \langle pc+1, cs, E, h', s\rangle}$$

$$\textsc{C-NewObj}\quad \frac{P(pc) = \textbf{newobj}\ T \qquad l \in \mathit{Loc}\ \text{fresh} \qquad h' = h[(l, f) \mapsto \bot]}{\langle pc, cs, E, h, s :: l\rangle \rightarrow \langle pc+1, cs, E, h', s\rangle}$$

$$\textsc{C-Ret}\quad \frac{P(pc) = \textbf{ret} \qquad st = (pc', E', s') \qquad pc'' = pc' + 1}{\langle pc, cs :: st, E, h, s :: v\rangle \rightarrow \langle pc'', cs, E', h, s' :: v\rangle}$$

$$\textsc{C-BrTrue}\quad \frac{P(pc) = \textbf{brtrue}\ i \qquad pc' = (v\ ?\ i : pc+1)}{\langle pc, cs, E, h, s :: v\rangle \rightarrow \langle pc', cs, E, h, s\rangle}$$

$$\textsc{C-Call}\quad \frac{P(pc) = \textbf{call}\ i \qquad st = (pc, E, s) \qquad E' = E[arg \mapsto v]}{\langle pc, cs, E, h, s :: v\rangle \rightarrow \langle i, cs :: st, E', h, \epsilon\rangle}$$

Fig. 12: Operational semantics of CIL

Appendix

A. A Primer on .NET Technologies

The .NET Framework is a managed execution environment for Windows providing a variety of services to its running applications. The framework consists of two major components: The Common Language Runtime (CLR), which is the virtual machine that handles running apps, and the .NET Framework Class Library (FCL), which provides a library of reusable code that developers can call from their applications. The FCL implements a collection of reusable types for user interfaces (e.g., XAML serializer), data access, web application development (e.g., JSON serializer), network communications (e.g., SOAP serializer) and other features. The .NET Framework implements the Common Language Infrastructure (CLI) specification, an ISO and Ecma standard that describes executable code and a runtime environment. Compilers for C# and F# generate code in the Common Intermediate Language (CIL) that can be executed in the CLI runtime. CIL is an object-oriented binary instruction set within the CLI specification. For our purposes, CIL provides a unified language for analyzing code from the .NET Framework and its applications in absence of source code.

The .NET Framework allows to dynamically instantiate arbitrary objects based on user-provided types and data. This is typically achieved via *reflection* which allows to examine the structure of types, create instances of types, and invoke methods on types, all based on the description of a type. Alternatively, the .NET Framework can instantiate an object at runtime via *dynamic code generation* by getting a pointer to a method and generating the CIL code of that method at runtime.