SECURITY ANALYSES FOR DETECTING DESERIALISATION VULNERABILITIES

A thesis presented in partial fulfilment of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

at Massey University, Palmerston North,

New Zealand.

Shawn Rasheed

2021

Contents

List of Tables

List of Figures

Abstract

An important task in software security is to identify potential vulnerabilities. Attackers exploit security vulnerabilities in systems to obtain confidential information, to breach system integrity and to make systems unavailable to legitimate users. In recent years, particularly 2012, there has been a rise in reported Java vulnerabilities. One type of vulnerability involves (de)serialisation, a commonly used feature to store objects or data structures to an external format and restore them. In 2015, a deserialisation vulnerability was reported involving Apache Commons Collections, a popular Java library, which affected numerous Java applications. Another major deserialisation-related vulnerability that affected 55% of Android devices was reported in 2015 [139]. Both of these vulnerabilities allowed arbitrary code execution on vulnerable systems by malicious users, a serious risk, and this came as a call for the Java community to issue patches to fix serialisation related vulnerabilities in both the Java Development Kit and libraries.

Despite attention to coding guidelines and defensive strategies, deserialisation remains a risky feature and a potential weakness in object-oriented applications. In fact, deserialisation related vulnerabilities (both denial-of-service and remote code execution) continue to be reported for Java applications. Further, deserialisation is a case of parsing where external data is parsed from their external representation to a program's internal data structures and hence, potentially similar vulnerabilities can be present in parsers for file formats and serialisation languages.

The problem is, given a software package, to detect either injection or denial-of-service vulnerabilities and propose strategies to prevent attacks that exploit them. The research reported in this thesis casts detecting deserialisation related vulnerabilities as a program analysis task. The goal is to automatically discover this class of vulnerabilities using program analysis techniques, and to experimentally evaluate the efficiency and effectiveness of the proposed methods on real-world software. We use multiple techniques to detect reachability to sensitive methods and taint analysis to detect if untrusted user-input can result in security violations.

Challenges in using program analysis for detecting deserialisation vulnerabilities

include addressing soundness issues in analysing dynamic features in Java (e.g., native code). Another hurdle is that available techniques mostly target the analysis of applications rather than library code.

In this thesis, we develop techniques to address soundness issues related to analysing Java code that uses serialisation, and we adapt dynamic techniques such as fuzzing to address precision issues in the results of our analysis. We also use the results from our analysis to study libraries in other languages, and check if they are vulnerable to deserialisation-type attacks. We then provide a discussion on mitigation measures for engineers to protect their software against such vulnerabilities.

In our experiments, we show that we can find unreported vulnerabilities in Java code; and how these vulnerabilities are also present in widely-used serialisers for popular languages such as JavaScript, PHP and Rust. In our study, we discovered previously unknown denial-of-service security bugs in applications/libraries that parse external data formats such as YAML, PDF and SVG.

Acknowledgements

All thanks be to God.

This was possible with the support and guidance of my supervisors Jens Dietrich, Amjed Tahir and Catherine McCartin. My utmost gratitude goes to them.

I am grateful to my mother and late father for everything. I don't have the words to thank my wife ♡ (and children) for being by my side during these years away from home and her family. I want to thank our families for their love and support. I thank Li Sui for his friendship. Thanks should also go to Massey University for the generous funding that let me pursue this.

Chapter 1

Introduction

An important task in software security is to identify potential vulnerabilities. Attackers abuse security vulnerabilities to gain unauthorised access to systems / confidential data, and to make systems unavailable to legitimate users. As software becomes ubiquitous in our lives and the economy, poor software quality and challenges in software development have lead to an increase in vulnerabilities and security attacks. According to the Consortium for Information & Software Quality (CISQ) [1] it is estimated that, by 2021, the global economy would bear a loss of USD6 trillion due to cyberattacks. In fact, 2020 ended with the very publicised SolarWinds Orion compromise[2].

As per the TIOBE index[3] for programming language popularity, Java has been at the top for the last two decades. The Java platform and programming language is heavily used in both server applications and mobile devices. Hence, it is desirable that applications written in Java are secure. However, in recent years, there has been a sharp rise in Java related vulnerabilities reported in 0-day attacks[4]. Two of the types of vulnerabilities involved in these attacks used invalid deserialisation [35].

Serialisation is a core feature in the Java language and it is commonly used in applications and libraries. Computer programs manipulate values. These values can be simple types like numbers, strings, and more complex types. Such complex types can be in the form of container types like lists, maps, compound data structures and user defined data types. Serialisation is the mechanism by which these values are written to an external stream and communicated to users and across programs and computer networks. The matching feature to read the data back from the stream is deserialisation. In Java, serialisation-based object storage and retrieval is used for lightweight persistence, communications over sockets, and Java Remote Method Invocation (RMI). Serialisation is widely used in services that enable distributed computing such as Java

[1] `https://it-cisq.org`
[2] https://www.solarwinds.com/securityadvisory
[3] `https://www.tiobe.com/tiobe-index/`
[4] An undisclosed software vulnerability that attackers can exploit, which has adverse effects.

Naming and Directory Interface (JNDI), Java Management Extensions (JMX), and Java Messaging (JMS). Distributed computing frameworks such (e.g., Apache Storm, Apache Spark) and web application frameworks (e.g., Spring) also use object serialisation.

In 2015, a deserialisation vulnerability was reported involving Apache Commons Collections, a popular Java library, which affected numerous Java applications. Another major deserialisation-related vulnerability that affected 55% of Android devices was found in the Android Platform in 2015 [139]. Both of these vulnerabilities allowed arbitrary code execution, a serious risk, and this came as a call for the Java community to issue patches to fix known deserialisation vulnerabilities in both the Java Development Kit (JDK) and libraries.

Since serialisation involves externalising data, in order for the data to be deserialised or the original state to be restored, the data must first be parsed. Hence serialisation vulnerabilities can manifest at the parser level or in the semantic phase of processing the data. For instance, XML parsing related DoS or injection attacks.

Despite regular patches, enhancement proposals[5], secure coding guidelines[6] and protective strategies, deserialisation remains a risky feature of the Java platform. Furthermore, vulnerability reports indicate that deserialisation based attacks are possible for other languages as well. Presently, detecting vulnerabilities that arise from the usage of deserialisation is done either manually or semi-automatically. Automation of the task will help in the improvement of software security and facilitate the safe and secure usage of this feature by making the process more effective and efficient.

1.1 Problem Statement

The research concerns the problem of deserialisation vulnerabilities and vulnerabilities in parsing data structures. This is important given the rise and potential impact of serialisation vulnerabilities in the Java ecosystem. There are multiple aspects to the problem: 1) detecting deserialisation vulnerabilities in Java libraries, 2) identifying these vulnerabilities in parsers written in other languages, and 3) preventing deserialisation-based attacks in applications.

There are two classes of attacks that exploit these vulnerabilities: remote code execution and denial-of-service (DoS). In a remote code execution attack, an attacker is able to exploit a vulnerability to execute their own code on the compromised system with the privileges of a legitimate user. This can impart the ability to steal confidential information or alter the system for further gain. The other type of attack, DoS, involves the attacker rendering a system inaccessible to legitimate users by either crashing the

[5] https://openjdk.java.net/jeps/290
[6] https://www.oracle.com/java/technologies/javase/seccodeguide.html

system or incapacitating it with requests for resources. We would like an automated solution to the problem of detecting vulnerabilities that can result in both these forms of attacks. Further, we are also interested in studying effective prevention measures against such attacks.

1.2 Research Objectives

The main goal of this research is to develop security analyses to detect serialisation vulnerabilities in Java software. A second goal is to propose mitigation measures that can detect potential serialisation attacks. Given a Java library as input, the security analysis would report likely vulnerabilities in the library. The proposed mitigation measure would detect serialisation attacks in running software. The objectives of the research serve to address gaps in static analysis tools and their capabilities in analysing applications with (de)serialisation features. There are multiple facets to their limitations: recall issues and precision. Furthermore, we address the understanding of a class of denial-of-service vulnerabilities, their prevalence in real world software and limitations in mitigation measures. More specifically, the objectives are:

- Identify existing gaps in vulnerability detection tools with respect to (de)serialisation.
- Examine existing fuzzing mechanisms and if they are an effective method to discover vulnerabilities in deserialisation.
- Determine effective ways to improve on the false positives issue in static analysis for vulnerability detection.
- Establish a formal model for a class of DoS vulnerabilities such that it can be used as a general model for vulnerability detection.
- Investigate if the studied DoS vulnerabilities generalise to other languages.
- Study effectiveness of existing mitigation measures and propose improvements.

1.3 Overview of Approach

The security analysis for detecting serialisation vulnerabilities is based on standard program analyses such as points-to analysis, call graphs and fuzzing. We extend existing techniques and frameworks to build an analysis for detecting vulnerabilities.

The question is how to construct effective program analyses to detect deserialisation vulnerabilities, which use dynamic language features that cannot be easily modelled [176]. Most static analyses, including points-to and call graph construction, are not

sound [105] (i.e., they may fail to model some program behaviour, which impacts the recall of these analyses). Much of the unsoundness is from dynamic program behaviour due to the use of programming features such as reflection, dynamic class loading, proxies and native methods [176] that are dynamic in nature. In some cases, soundness can be recovered by heavily over-approximating dynamic program behaviour[167]. However, such analyses can be practically useless due to high imprecision [106] or poor scalability.

1.4 Contributions

The main contributions of our work are: the development of security analyses, a proposal for mitigation through detection of attacks at runtime, and a systematic study of non-Java serialisers for vulnerabilities. During the course of this work we reported multiple vulnerabilities (listed in Chapter 9, Sec. 9.2) in widely used software. Specific contributions:

Vulnerabilities. We studied parsers for external formats and data serialisation languages in Java. We reported 11 issues and obtained four CVEs.
Systematic study of parsers. We studied 14 libraries for the YAML data serialisation language across 10 different programming languages. We discovered and reported seven previously unknown vulnerabilities.
Reporting. False positives is an issue in the results of static program analyses. For the particular case of statically analysing libraries for serialisation vulnerabilities, we propose a technique using fuzzing to eliminate false positives and reduce fuzzing time by using the results of the static analysis to scope fuzzing.
Mitigation. We propose two techniques for sanitising input as protection against a class of DoS vulnerabilities for deserialisers and file format parsers. One is based on the structure of the input. The other technique, which is specific to Java, is based on dynamically checking the type of the input structure.

1.4.1 Publications from Research

The research contributed to the following publications.

- Chapter 6: *"Evil Pickles: DoS attacks based on Object-Graph Engineering"*. Dietrich, J. and Jezek, K. and **Rasheed, S.** and Tahir, A. and Potanin, A. The European Conference on Object-Oriented Programming (2017).

- Chapters 6 and 8: *"Laughter in the Wild: A Study into DoS Vulnerabilities in YAML Libraries"*. **Rasheed, S.** and Dietrich, J. and Tahir, A. The 18th

IEEE International Conference on Trust, Security and Privacy in Computing and Communications (2019).

- Chapter 7: *"A Hybrid Analysis to Detect Java Serialisation Vulnerabilities"*. **Rasheed, S.** and Dietrich, J. The 35th IEEE/ACM International Conference on Automated Software Engineering (2020).

1.5 Outline

We present the background on software security and program analysis in Chapter 2. Chapter 3 contains a survey on serialisation and serialisation-related vulnerabilities. Chapter 4 presents an account of the research methodology that was followed for the studies. In Chapter 5, we propose a static analysis for detecting DoS vulnerabilities in format parsers. In Chapter 6, we discuss serialisation vulnerabilities in non-Java languages. Chapter 7 proposes a hybrid analysis for detecting injection vulnerabilities related to serialisation.

In Chapter 8, we present mitigation measures against DoS vulnerabilities. Finally, concluding remarks and directions for future work is presented in Chapter 9.

Chapter 2

Background

2.1 Software security

In software development the attribute that developers primarily focus on is the correct functionality of the software they produce [113]. Correctness broadly means demonstrability that the software behaves as expected under normal circumstances. However, software can be determined to be secure only when it also prevents undesirable behaviour in expected and unexpected situations.

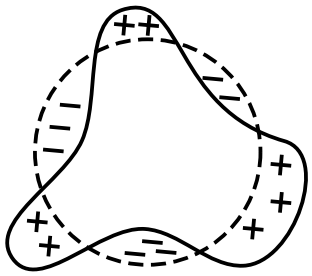


Figure 2.1: Implementation vs needs/design

Figure 2.1 depicts the nature of the problem as described by Felderer et al. [66]. Design flaws or bugs (implementation defects) are the result of unimplemented behaviour (the negative signs) and unintended side-effects or incorrectly implemented behaviour (the positive signs). Faults in security mechanisms are the missing functionality (e.g., non-sanitised inputs to security sensitive operations, missing permission checks that result in information leaks). Vulnerabilities can also occur in the shape outside the design. The perfectly secure system would be where the implementation completely overlaps the intended design (and perhaps also unspecified safety/security properties that will be of concern to users) but this is usually not the case and mismatches result

in insecure software that is the target of attacks. In order to be considered safe and secure, software must address three aspects: data confidentiality, data integrity and availability, by either preventing or mitigating attacks upon them.

Confidentiality is ensured when a system makes it impossible for attackers to obtain sensitive information. In secrecy attacks, the attacker's attempted objective will be to steal confidential information from the system. This information might be passwords, records, electronic mail, business or customer data.

Integrity entails the ability of a system to prevent unauthorised changes to the system that results in misuse of the system. In integrity attacks the attacker's goal is to modify the components of a system illegally. The modified components might be system configuration, logs or business data on a system. The motive behind such attacks could be to obtain full access to a system or change information to the attacker's benefit.

Availability concerns the ability of a system to withstand attacks that may result in compromising the system so that its services are not available to legitimate users. Attacks that disrupt the normal functioning of a system are commonly known as DoS attacks or space and time attacks. These attacks exhaust space (memory, stack or disk) or time (processor). DoS attacks can be network based, where the attacker bombards a system with a large number of requests, or an application crashes or its performancce is degraded due to a software defect that is exploited by an attacker.

2.1.1 Java Security

Java is defined as a managed and type-safe language built with a focus on security [114]. Language features that enhance security include automatic memory management, garbage collection and bounds checking on strings and arrays [114]. Emphasis on memory-safety eliminates the types of low-level memory exploits popular with languages such as C (e.g., buffer overflows), although these attacks may still be possible at the native interface level, which could be implemented in a language like C.

Applications are run on the Java Virtual Machine (JVM), which places additional security constraints on program execution. The Java runtime (JRE) loads library and application code using the class loading mechanism where Java bytecode is verified before loading, and application classes are isolated from restricted packages in the Java class library. Additionally, the sandbox model in the JVM isolates the execution of untrusted code from accessing sensitive system resources. Access to sensitive system functionality is controlled by the security manager, which checks if the calling classes have permission to access the operation. Despite the security features offered by the language and the Java platform, defects in trusted code can be the cause for vulnerabilities. The parts of the Java security model are described in more detail below.

a. Class loading

Class loading is the mechanism by which Java bytecodes are read into the JVM and converted to class definitions for execution. A class loader works with the security manager and access controller to establish most of the protection offered by the Java sandbox, which is essential when classes are loaded from untrusted locations over the network where the resources they can have access to must be limited. The tasks of the class loader are: to return a class object if it has already been loaded by the particular class loader; security manager checks if the program has permission to access the requested class; the class is read into an array of bytes and it is verified by the bytecode verifier; classes that are immediately referred to by the loaded class are also found by the class loader.

b. Security Manager

The installed security manager determines if particular operations must be permitted or rejected. If a Java program desires to access the file system or make a network connection it must request for permission from the security manager.

c. Stack-based Access Control

The JVM's security architecture implements access control by utilising information available on the call stack. If a security manager is installed, and a request is issued by user code for access to a resource or action that is restricted then a permission check is performed. If calling the `checkPermission()` method on the security manager results in an exception being thrown the request is not allowed. The issue is deciding which permissions are in effect when the request is issued as there may be multiple sets of permissions in use at the time (system, user and library code). The JVM makes this decision by inspecting the call stack and computing the intersection of the protection domain of every class found. If the permission is in this intersection, then the request is allowed otherwise an exception is thrown.

d. Information Hiding

Information hiding is an important feature of the Java platform. Access to private fields and methods are modularly restricted. If access was gained it would be possible for user code to manipulate the integrity of system classes such as the security manager to avoid security checks. However, language features such as reflection allow users to bypass access restrictions.

e. Type Safety

Another aspect of the security model is type safety. The system keeps tracks of the types of objects during compile and runtime and the kinds of operations that can be performed on the particular type of the object. Type confusion would allow performing an operation on an object that is not permitted on it, which may have security implications.

2.1.2 Security-sensitive Language Features

The JRE offers special features to programmers that can be abused to subvert the security model [89]. Two of these features are reflection and deserialisation. The reflection API can be used to write reflective code that can dynamically (during runtime), create objects, access an object's fields and call its methods. Listing 2.1 shows the use of reflection to load a class, create an object instance and invoke a method on it.

```
1 String className = ...;
2 Class c = Class.forName(className);
3 Object o = c.newInstance();
4 String methodName = ...;
5 Method m = c.getMethod(methodName,...);
6 m.invoke(o, ...);
```

Listing 2.1: Use of reflection APIs

Binary object serialisation in Java breaks information hiding for serialisable classes. Even if an object has fields that are declared private, serialisation would expose the data in the stream. Security lapses in deserialisation are discussed in more detail in Chapter 3.

2.2 Vulnerabilities

2.2.1 DoS Vulnerabilities

A DoS attack is a threat to the security of computer networks, as it attempts to make the services of a computer system unavailable to its users. Common DoS attacks work by exhausting the resources of a server to the point that it is not available for use. DoS attacks can also target client-side applications. A number of vulnerabilities in software can expose a system to DoS attacks. Such attacks can be broadly categorised into network-based and host-based attacks [84]. In this thesis, as far as DoS attacks are concerned, we focus on the latter, and on application-layer vulnerability attacks also referred to as semantic attacks [4]. Network-based attacks are not in the scope of this thesis.

In Java, DoS attacks can either target memory (resulting in memory exhaustion), or cause worst-case algorithmic complexity behaviour that induces indefinitely long computations resulting in service unavailability. Widely used data structures have efficient average time complexity but they can exhibit poor behaviour on certain inputs. Examples are hash tables that degenerate to lists, from constant time to linear time lookups, on inputs with hash collisions. An attacker can take advantage of such performance issues in a program to execute a DoS attack [47]. Two of these vulnerabilities are presented by Polesovsky [150]. A nested set of arrays is crafted with each array having a maximum possible size set to the maximum integer value. Deserialising this object exhausts heap space as it allocates large chunks of memory for each object. The second payload that targets Java 1.7 uses hash collisions, by creating a `HashMap` or `Hashtable` with the initial capacity set to the load factor of the `Hashtable` resulting in a degenerated hash table that uses a single bucket to store all items. There are a few other serialisation-based attacks that exploit worst-case algorithmic complexity behaviour such as *SerialDoS* (described in Chapter 3) and an exploit that uses a serialised regular expression pattern object [51]. The regular expression exploit as described by Schönefeld [163], is a result of doubling compile time for each group in a pattern, and deserialising a pattern with fewer than a hundred groups can take several hundred years to compile.

Billion laughs is a well-known DoS attack that targets XML parsers [39]. It consists of a Document Type Definition (DTD) part, which describes the structure/grammar of the document within itself, that causes parsers to consume the processor or memory to the extent that it results in a DoS. The inline DTD defines a list of nested XML entities where each entitys definition contains references to the preceding entity definition. The expansion of the entity defined at the bottom results in an exponentially large string that in effect causes the service to degrade or fail. Some parsers protect against this attack by introducing a threshold for entity references within a document.

Another variant of the attack, known as the *quadratic blowup* [44] cannot be avoided using a simple threshold. *Quadratic blowup* consists of an entity definition with a single large string that can be referenced a few times (a quadratic growth) to cause a performance blow up when parsing the document.

Späth et al. [171] describe recursively defined entities, which reference each other in their definitions. Even though the XML specification forbids such definitions, some parsers are susceptible to DoS attacks via such XML documents that put the parser in an infinite loop. A similar DoS vulnerability that exploits PDF file document outlines, which is implemented as a doubly-linked list structure within the document, is discussed in [62], where a badly-formed outline with cycles is demonstrated to cause DoS in PDF clients.

2.2.2 Code Injection and Information Leaks

It is common for programs to obtain and process input from untrusted sources. Examples are a user entering input, or a web form submission. Untrusted input is a common starting point for a number of software vulnerabilities. Most notable among which are code injection vulnerabilities such as SQL injection, cross-site scripting (XSS) attacks and shellcode injection. Input validation or sanitisation is the usual way to guard against untrusted input. Vulnerabilities that are the result of unchecked processing of untrusted input are classified under CWE-20 in the Common Vulnerabilities and Exposures list (CVE). Most security vulnerabilities[1] can be classified as information-flow vulnerabilities.

In an information-flow based security model, data that is manipulated by a program can have security levels associated, which in the simplest case would be *high* or *low* [149]. The non-interference model [76] for security states that low-level security behaviour of a program must not be affected by high-level security data. In other words, non-interference is satisfied if a programs public outputs do not depend on the values of its secret inputs. The levels *high* and *low* can be interpreted as confidential and public respectively. The non-inteference criterion can be verified if there is no flow from a *high* to *low* level (without passing a declassification check). For integrity, these levels can be viewed as untrusted and trusted, and integrity can be verified if there is no flow from an untrusted to a trusted level (without endorsement, more commonly referred to as sanitisation in practice). Vulnerabilities due to unvalidated input such as injection can be viewed as violations of integrity, and untrusted input data is referred to as *tainted*.

Low-level memory attacks in languages like C, such as buffer overflows, format string vulnerabilities are information-flow vulnerabilities. They can be exploited for code injection attacks. These can be defended against with techniques like data execution prevention and mechanisms like stack canaries (integer value placed before the stack pointer) to check if the stack has been overwritten by untrusted input. These advances have not completely halted low-level exploits which have evolved into more complex code reuse attacks where attackers do not inject code but reuse existing library code to put together a remote code execution attack. The first attacks of these kinds were return-into-libc where attackers can arrange items on the stack so that an arbitrary function with parameters can be called forming an attack [166]. In this work, Shacham shows that code reuse attacks are not limited to functions in libc but can be strung together with sequences of code within the library. These blocks of instructions are called gadgets and can be chained together as *gadget chains* that repurpose existing code for attacks. Attacks of this form have come to be known as return-oriented programming (ROP) attacks.

[1] https://owasp.org/www-project-top-ten/

For scripting languages (e.g., PHP) and Java, variants of code injection are possible. Instances are SQL injection, expression language injection and reflection injection vulnerabilities. Dahse et al [56] show how ROP type attacks are possible with PHP serialisation. The premise is that objects have properties which are themselves objects that can be crafted as gadget chains. On deserialisation, magic methods (methods that are executed automatically for deserialisation) cause the execution of the complete gadget chain. The mechanics of how this works for Java deserialisation is discussed in Chapter 3, Section 3.6.1.

2.3 Language-theoretic Security for Preventing Vulnerabilities

One of the approaches to preventing these vulnerabilities and resulting attacks is to use techniques from language-theoretic security[2]. Sassaman et al. [159] note that parsing input from untrusted sources constitutes the basis for most attacks on computer systems. If parsing and subsequent input validation is not well-designed and validation code is strewn across the program in a way that may not adhere to a programmer's assumptions about safety and security, it can result in exploitable vulnerabilities. The idea is that such code represents a computational model embedded in the vulnerable application, referred to as weird machines, that can be programmed by an attacker with crafted input [28]. The language-theoretic approach to the problem is to treat input validation as a language recognition problem. Valid inputs are modelled as a formal language and input validation is presented as checking if the given input belongs in the language which defines valid inputs. Any input that cannot be recognized is rejected without any further processing. Sassman et al. [159] discuss how this applies to injection attacks (e.g., SQL injection, command injection for interpreters) and low-level attacks such as buffer-overflow. The mitigation measures that we discuss for deserialisation (e.g., using inclusion and exclusion lists) in Chapter 8 are presented as techniques to safeguard input handling.

2.4 Detection of Vulnerabilities

Code review and automated program analysis are used for detecting software vulnerabilities. In a comparison of these techniques, Edmunson et al. [61] have shown that code walkthrough is an expensive and unpredictable process. The study of developers conducting a security review of a small web application revealed that not all of them could detect all the vulnerabilities present in the software. Furthermore, it was found

[2] http://langsec.org/

that developer experience was not a useful predictor of the ability to discover more vulnerabilities or the developers' accuracy or effectiveness at the task. Thus, this lack of efficiency in code walkthrough and the growth of complexity and multitude of reported vulnerabilities have given rise to an abundance in research and tools to automatically find vulnerabilities.

Program analysis (automated analysis of code), initially developed as part of compiler technology to produce efficient code, is an efficient method to mitigate the risk of malicious or defective code [123]. Program analyses can be classified as static analyses, dynamic analyses or hybrid analyses. Table 2.1 illustrates the differences in the types of analysis. Ernst [63] has discussed the hybrid approach to combine dynamic and static analysis techniques for better results. One suggested way is to use the output of one analysis as input for the other. This approach has been explored in tools such as DSDCrasher [50].

Table 2.1: Static vs dynamic analysis

Static analysis	**Dynamic analysis**
Abstract	Concrete
Slow if precise	Slow if exhaustive
Approximates behaviour	Exact behaviour
Sound (not always) and generalises	Unsound

Apart from program analyses to detect programming flaws, there has been work on preventing vulnerabilities using programming language features. In this preventative approach to the problem of software security, the methods fall under the umbrella of *language-based security* introduced in [101]. In-lined reference monitors to enforce security policies, proof carrying code to establish the integrity of code, and type systems to enforce information flow policies are some of the ideas in this area [161]. In this category, JFlow [121] is a popular extension to Java that enables the language to statically check for information flow vulnerabilities before they occur in the compiled software. Checker framework [136] is another such tool for Java that supports pluggable types to ensure information-flow properties hold at compile time. The code must be annotated with type information for the checker to perform the check.

2.4.1 Static Analysis

Static analysis models program behaviour without executing the program. It produces results by examining code structure, sequence of statements, how data flows inter and intra procedurally through variables and function calls through the modelled execution.

Static analysis is appealing because of its ability to model all behaviour of a program or in other words, complete coverage. However, Rice's Theorem entails that the problem of answering non-trivial and interesting questions about the semantic properties of a program is undecidable [118]. It is still possible to obtain useful results from a static analysis for tasks such as finding bugs, but the analysis can only obtain approximate answers. Over-approximation in static analysis is problematic as the number of false positives can overwhelm developers [24]. Furthermore, static analysis finds it practically difficult to capture program behaviour due to the use of dynamic features in certain languages without an extreme loss in precision. Attempts to gain precision with extra modelling, results in computational expense which is prohibitive for large applications. Tradeoffs can be achieved by sacrificing soundness for faster analyses [106]. However, tradeoffs of this form are not deemed safe for security analyses as the requirement is to catch a vulnerability if there is any.

Pistoia et al. [149] surveys static analysis methods used for finding security vulnerabilities in software. They cover three areas of security bugs: access-control, information-flow, which concerns information leakage and application-programming-interface conformance. They discuss how static analysis can be used for detecting violation of policies for integrity and confidentiality and how it can be used to verify for correct use of security libraries and interfaces.

In this research we use Doop [29], which encodes static analyses declaratively in the Datalog [3] language. This facilitates extending Doop with customised analyses. In addition to Doop, Soot [188] and WALA [32] are widely used static analysis frameworks for Java, which implement the static analyses listed below.

a. Control Flow / Call Graphs

Control flow graphs (CFG) are an abstract program representation for flow-sensitive static analyses. In CFGs vertices are program statements and the directed edges represent flows between statements. Interprocedural control flow across methods in the program can be represented using call graphs.

Call graph construction for object-oriented languages has been discussed by Grove et al. [83] and Ryder [158]. It is a useful representation of control and data flow in a program and its construction is a common step in performing any static analyses, especially any inter-procedural analysis. In its most basic form a call graph models caller and callee relationships in a program as a directed graph consisting of vertices that represent method names (constructors, static or instance methods in object-oriented languages) and edges connecting methods that represent call sites within the caller method with the target being the vertex that is the callee method. Listing 2.2 shows Java code where method `foo` calls method `bar`. The corresponding call graph is shown

in Figure 2.2

```
1   class A {
2     public foo(T t){
3       B b = new B();
4       b.bar(t);
5     }
6   }
7   class B {
8     public void bar(T t) {
9       t.baz();
10    }
11  }
```

Listing 2.2: Call-graph sample code

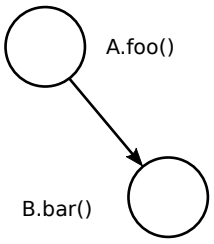


Figure 2.2: Basic call graph

In object-oriented languages with dynamic dispatch, it is important to obtain a call graph that is a conservative approximation of all methods that a virtual call site may dynamically dispatch to. For instance the call site `t.baz()` in the listing. The target method could belong to any subtype of the class `T`. [184] and [180] discuss various call graph algorithms and the extent to which they achieve this. The standard method to devirtualize calls is using Class Hierarchy Analysis (CHA). If at a call site, a receiver has a declared type `T`, then the possible runtime types of the receiver are subclasses of its declared type and, if it declares an interface, the set of all classes that implement that interface or sub-interfaces and all subclasses of them.
Rapid Type Analysis (RTA) [20] eliminates extraneous edges from the extremely conservative call graph constructed using CHA by only considering types that have been instantiated in the program. Variable Type Analysis (VTA) [180] adds a fine-grained constraint where it examines the concrete type of the receiver variable for a more precise approximation. Note that this requires another analysis, points-to or alias analysis to explore what object a variable may point to, which is usually achieved by a simultaneous points-to analysis known as "call-graph construction on-the-fly" [169].

Call graphs can be used for reachability analysis to determine if there is a path that connects two program execution points in the graph. For example, a query might

be if calling a method `f()` may result in the eventual invocation of another method `g()`. After obtaining the CFG of a program, standard graph algorithms can be used for performing this analysis.

b. Dataflow Analysis

Dataflow frameworks can be used to compute properties such as taint reachability, constant propagation, live variables at program points [118]. The abstract computation is based on lattices, which models program state, and transfer functions which compute how state is transformed at each program statement for the particular analysis. At the interprocedural level, there are two approaches for dataflow analysis. The call string approach and the functional approach. The functional approach, based on the IFDS framework [155], is used in tools like FlowDroid [16] and FlowTwist [103].

c. Points-to Analysis

Points-to analysis [169] computes the set of objects that a pointer/reference variable might refer to at runtime. It is an approximation which is fundamental in many areas of static analysis. It is useful in on-the-fly call graph construction for object-oriented languages. The precision of points-to information can in turn result in more precise call graphs. The results of points-to analyses can also be used to check for aliasing (a client analysis for this feature is taint analysis), i.e., if two variables point to the same object.

c. Static Taint Analysis

In Section 2.2.2 we discussed information-flow based vulnerabilities and tainted input. Static taint analysis is an approach to check for information-flow properties. The goal of a taint analysis is to establish if tainted input flows from a source to a sink in an application without going through a sanitisation check. The source can be a point of input for the application (for a web application, the request handler) and the sink, a security-sensitive method such as a one that accesses an external resource (e.g. database, filesystem or network) or performs a critical task.

Taint analysis has two variants: dataflow-based, which is concerned with tracking the flow of values in a program, or type system based. Tripp et al. in [186] claim that both approaches have the disadvantage of producing too many false positives, which limits the usability and applicability of the techniques in software development. Work to reduce false positives results in unsoundness that causes false negatives to rise and attempts to improve precision results in expensive analyses that do not scale due to the additional computational cost of modelling more aspects of the program. They propose an alternative approach that is a demand driven flow analysis which scales as

a result of its non-eager approach. Two state-of-the-art general purpose static taint analysis frameworks are FlowDroid [16] and Doop [81]. Unlike FlowDroid where the dataflow-based taint analysis is a client of the pointer analysis, Doop combines the computation of points-to with information-flow.

Livshit and Lam describe a taint-based static analysis for detecting Java vulnerabilities in [106]. The tool uses context-sensitive pointer analysis to improve precision while maintaining the soundness of the results. It uses PQL to specify vulnerability patterns common in web applications which are used to generate static analysers to find the matching vulnerabilities. The static analysers track object propagation from the user-defined sources to sinks. The tool detected 29 undiscovered vulnerabilities in nine large popular open-source web applications.

Since detection of most vulnerabilities can be framed as information-flow problems, taint analysis as information-flow analysis has been used for identifying code injection attack vulnerabilities for Java web applications [185] [186] and static analysis has also been utilised in security analysis for caller-sensitive method vulnerabilities [35]. The application of taint-analysis to "object injection" using serialisation has also been studied for PHP [56].

d. Program Slicing

Program slicing [191] is a technique to simplify a program by selecting a portion/slice of the program that is relevant to the computation involving a set of variables or a particular point in the program. Program slices can be backward slices or forward slices: a backward slice identifies program statements that affect the computation of a selected statement; a forward slice identifies statements affected by a selected statement. Tripp et al. [185] have used program slicing for taint analysis of web applications. The technique used in the paper is to simplify the computation of source to sink flow of untrusted data by obtaining a slice of the program that is data-dependent on the source.

e. More Precise Call-Graphs and Points-To Analysis

Call graphs as described earlier achieve an over-approximation of the actual call structure of the possible executions of a program, which can be made more precise by making the analysis sensitive to various aspects of program structure and execution. In interprocedural analysis this also applies to other static analyses such as dataflow and points-to. The types of sensitivity for an analysis are:

Flow sensitivity. Flow sensitivity takes the order of statements in the program into account in an analysis. For example, consider a variable of a particular class and two objects instantiated in the program that belong to two different subtypes of that class. If the variable is assigned one of the objects and a method call is invoked on the object,

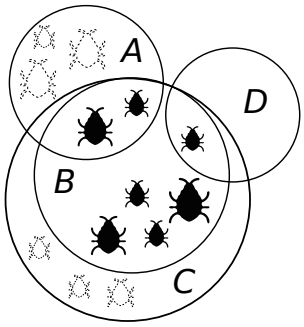


Figure 2.3: Detecting bugs using analysis

this call will be devirtualized to the type of the object it had been last assigned before the call.

Path sensitivity. A path sensitive analysis distinguishes between execution paths over different branches.

Field sensitivity. In a field sensitive points-to analysis it tracks values assigned to fields of an instance separately.

Context sensitivity. In context-sensitive analysis the calling context is considered in interprocedural control flow. For example, if a method call takes an object as an argument whose type is a subclass of the declared parameter type, a context-sensitive analysis would not merge the arguments of the passed object at each call site. If the calling context is not taken into account, the analysis would be imprecise due to merging of information across call sites for the method.

Object sensitivity. Object sensitivity is a type of context-sensitivity [115] where the receiver of a method call is taken into account.

2.4.2 Challenges for Static Analysis

Soundness and precision are two of the key challenges for static analysis [106]. Soundness meaning that, if a bug is there it must be reported and precision meaning that, if a bug is reported it must be present. Figure 2.3 illustrates the problem with static analyses. The filled in bugs are correctly identified vulnerabilities (true positives) and the bugs with outlines are falsely reported vulnerabilities (false positives). The ideal analysis would report the results in circle B: all vulnerabilities and only correctly identified vulnerabilities. With dynamic analysis (circle D) one obtains precise results but these are usually incomplete in the sense that not all bugs are reported. An analysis that is unsound (circle A) may report false positives but, more importantly, does not report all bugs. A sound but imprecise analysis (circle C) reports all bugs but also reports false positives.

Reif et al. [154] and Allen et al. [6] discuss the issues in constructing call graphs for libraries. Problems arise primarily due to the lack of a specific entry point for analysis in the absence of information about the application using the library, which necessitates treating all public entry methods as possible entry points. Another case is the open world type system of Java which makes results unsound.

Java's type system is open-world due to features such as dynamic class loading (could be over the network). It is impossible to determine the set of classes that will be available to a running program that uses these features, which results in unsound analysis results. Other dynamic language features such as reflection and dynamic proxies are notorious hindrances to obtaining sound analysis results [176].

2.4.3 Dynamic Analysis

In contrast to static analysis, which models program behaviour without executing the program, dynamic analysis involves concrete execution of a program to determine its behaviour. As these analyses only partially exercise the program, this technique is unsound [63]. Another issue is input generation for program execution, which asks what input produces the desired behaviour that is the target of an analysis. This can be particularly difficult if the inputs are constructed manually for execution. From a program testing perspective (a dynamic analysis), to determine if a property holds for a program by executing it, we require an input and a test oracle. A test oracle is a procedure that differentiates correct and incorrect behaviour. It can be difficult to manually construct inputs to sufficiently cover program executions that can uncover defects or faults. Hence, there is an increasing trend towards automatically generating tests instead of manually specifying them [9].

The dominant approaches broadly fall under random testing, dynamic symbolic execution [75] and search-based testing. Whereas basic testing involves concrete input, in property-based testing and generative testing, inputs are generated during testing. Input generation can either be random (guided by feedback or a search-heuristic) or input selection can be informed by a model. There is a vast amount of research into test generation and we touch on techniques for the Java language. Fuzzing and symbolic execution, in particular, are popular approaches that have yielded useful results in software security.

Fuzzing

Fuzzing, widely used in security testing, uses random test input generation to detect undesirable program behaviour such as unhandled errors, non-termination, and resource exhaustion. Input generation can either be generational, using a provided input model, or achieved by mutating an initial set of inputs to derive additional

inputs. Fuzzers can go further when informed by input-structure (grammar-based) or/and program-structure. Fuzzers that are aware of program-structure are described as whitebox fuzzers. Fuzzers are blackbox if they are not aware of program-structure. Black box fuzzing is inexpensive but it can be ineffective when it comes to exploring deep into a program's state space. Whitebox fuzzers can use program analysis for more effective test input generation.

Fuzzing was first explored by Miller et al. [116] to study the robustness of UNIX programs using an inexpensive mechanism to discover bugs and improve the reliability of widely used programs. The setup consisted of two tools, fuzz and ptyjig. Fuzz generates a constant stream of random characters to be sent to programs and ptyjig is used for sending characters to programs that require interactive input. If the program crashes the core dump and input file are saved for later analysis. Blackbox fuzzing continues to be effective in fuzzing file formats against parsers and for network protocol fuzzing. A blackbox approach that is conceptually similar to fuzzing is penetration testing which aims to determine if an application has vulnerabilities that can be exploited to compromise the application, data or the application environment. It is a popular blackbox approach for security testing networked computers for known vulnerabilities [22] evaluates blackbox web vulnerability testing tools.

Godefroid et al. leverage their directed random testing DART tool in SAGE [75], a whitebox fuzzing system that has been very successful in finding security vulnerabilities at Microsoft in some of their flagship products (PowerPoint, Web browsers...etc) and SAGE continues to be used daily for detecting security bugs. JFuzz [91] is a concolic execution based fuzzer like SAGE for Java. Documentation reveals it is a input file fuzzer like SAGE and not suitable for unit test type fuzzing of an object-oriented program. Another input file fuzzer that uses taint-based guiding is described by Ganesh et al. [72]. It fuzzes parts of the input file that are most likely to influence the program's execution by means of taint tracing.

Random Testing

Random testing has proven to be an effective and, at the same time, inexpensive method to discover faults in programs. For object-oriented programs random testing is different from fuzzing as fuzzing involves random data and object-oriented randomisation requires random primitives and random method invocations to mutate objects. A number of tools are available for random testing of Java programs. The oracle problem (heuristics, contracts and hardcoded assertions are some of the approaches), test case minimisation, feedback and guiding are areas of research. Challenges in the area are low coverage and efficiency.

Random Testing without Guidance

JCrasher [48] examines the type information of a program and produces a datalog graph to transitively compute inputs for Java methods. To test a method, the approach looks at its parameters, searches possible ways to compute the inputs for parameters and generates statements that select a random combination of inputs that satisfies the parameters. This computation is random and it generates sequences of statements for the inputs as JUnit test cases. It uses heuristics to add assertions for test cases it determines uncover actual errors.

Eclat [131] takes an existing test case suite and the corresponding application and diversifies the test case suite by adding new test cases that have randomly selected inputs. Eclat infers invariants using the Daikon [64] tool during the execution of the test suite and classifies generated test cases with respect to preconditions inferred in the pre-step. Jtest is a commercial random testing tool for Java that produces test cases based on user provided pre-condition/post-condition contracts.

Jartege [130] generates tests based on Java Modelling Language (JML) specifications written by the user. RUTE-J [10] is a similar tool but adds the capability to execute all generated test cases together and minimises the test cases.

Randoop [132] is one of the more popular and effective random testing tools for Java. It has been ported to .NET and is in the same class as Microsoft PEX [183] for random unit test generation. Its novelty is that it incorporates feedback from executing test inputs as they are generated. In brief, the technique works by randomly selecting a method call and finding arguments for the call with previously built inputs. When an input is built, it is executed and checked against a set of contracts and filters. The result determines if the input is redundant, illegal or breaks a contract and if it is useful for generating more tests. In experiments feedback-directed random tests are shown to outperform systematic and undirected random tests with regard to coverage and error detection.

Random Testing with Guidance

Palulu [14] improves on Randoop by guiding the random test generation with a model. The model is the result of a dynamic analysis which involves executing the program and observing the methods. Multiple executions of the class under test (CUT) are done and the result is combined as a graph. The random execution picks a node from this graph which is a point in execution and randomly picks an edge to the next node. The edge represents a method and parameters. Test case generation seeks variety in test cases by randomising these parameters.

Check 'n' Crash [49] uses the Extended Static Checker for Java (ESC/Java), a static annotation-based program checker, [67] to statically test for bugs and then compiles the

reports to actual test cases by applying constraint solving to the ESC/Java annotations and filters out reports that cannot reproduce the bug when executed. Hence it uses a static and then dynamic technique to improve the precision of detecting bugs. However, it is limited to the types of bugs that ESC/Java finds in Java methods, which is the the result of an analysis that is both unsound and imprecise.

DSDCrasher [50] improves on Check 'n' Crash. Like its predecessor, it is a hybrid analysis tool that combines static and dynamic program analysis. It consists of dynamic inference using Daikon [64] to detect program invariants, static analysis within the restricted input domain to explore paths and finally automatic generation of test cases that focus on predicting the results of the static analysis.

Ma et al. presented GRT [110] that uses constant mining to improve the set of values used in parameters for random method invocations during test generation. It makes additional efficiency gains by only considering methods that have a side-effect. This filtration is done by an impurity analysis. Ma et al reported that the tool shows better results in comparison to Randoop and Evosuite. However, the tool is not publicly available.

Other hybrid approaches include the tool from Pradel et al. [151] which uses protocol mining - discovering the order API methods are called - to check randomly generated tests against the mined protocols. Only tests that reveal safe usage of the APIs (where safe is defined as protocol adherence) are filtered out for further observation thus eliminating false positives. Babic et al. [19] apply a hybrid approach to directed test generation for binaries. Their application is limited to finding buffer-overflow vulnerabilities and does not scale to large programs due to the costly but precise static analysis.

Constraint-based Test Generation

Symbolic execution was first presented in [99] as a method to test and debug programs by providing symbols as input instead of values. A program is then consecutively interpreted resulting in expressions over the input symbols. In combination with expressions that are path conditions over which a program's execution branches and a constraint solver, symbolic execution-driven testing can discover concrete inputs that can lead the execution of a program down selected branches. These inputs can then be used for testing with the goal of increasing test coverage. There are limits to classic symbolic execution such as the scalability issues with path explosion and constraints (e.g. floating point inputs, native functions) that constraint solvers are not equipped to cope with.

It is known that simple random testing does not get deep into a program's state space. Directed Random Testing [74] (DART) attempts to address this via what has

come to be known as concolic testing, a combination of dynamically executing a program and using symbolic execution to solve path constraints which are used to solve parts of the input towards exploring different execution paths. DART consists of three components: interface extraction, generation of a test driver for random testing through the interface, and dynamic test generation to guide execution along different program paths. The first part is achieved through static code parsing to discover functions to be tested. The driver provides random input for the function interface and executes it. The generated driver can be modified with pre-conditions and post-conditions as assertions. During each execution of the function, an input vector is generated, which will be used as input for the next execution. The execution continues until all paths have been explored or a defect is found. Randomisation is used where solving constraints is difficult. In contrast to random testing, using concolic testing allows for bugs to be found more quickly and the state space to be explored more efficiently.

Search-based Testing

In contrast to constraint-based techniques like symbolic execution for generating test inputs, search-based tools use meta-heuristic search techniques to optimise the generation of test suites for a specified criteria (e.g., code coverage). EvoSuite [68] is one of the tools in this category. It uses a genetic algorithm to evolve an initial set of random test cases with the goal of maximising coverage. The search operators it applies are selection, mutation and crossover to evolve the tests. The evolution is guided by a function based on a criterion for which the default is code coverage. This fitness function can be modified to fit custom requirements. When the search completes, the suite with the highest code coverage is minimised according to the coverage goals and regression test assertions are added. Finally, EvoSuite executes the tests to confirm they are syntactically correct.

Combinatorial Testing

Combinatorial design theory has been applied to testing where tests are composed of all pairwise, triple, n-way combinations of test parameters [38]. The key contention is that pairwise coverage sufficiently matches code coverage for the system under test (SUT). [122] briefly describes the general approach: consider the SUT has n parameters, which may be inputs, configuration settings, external events or user interactions. Next, a set of interaction relations (set of pairs of parameters over the domain $(1..n) \times (1..n)$) is defined that models potentially problematic interaction between the parameters. This information must be obtained from the source code or domain experts with knowledge of the parameters, for example, where they are external and may not be available in the code. If $c1, c2$ is in the set of relations it denotes that $c1$ and $c2$ interact in a way that

may produce failures in the SUT. An issue with combinatorial testing is if the input domain is large there is an explosion in the number of combinations. One approach is to partition the parameter space and select inputs that are representative for each partition. This is reminiscent of equivalence class testing.

Adaptive Random Testing (ART)

The idea behind adaptive random testing is to make random testing more efficient. The technique is to distribute test cases over the input space instead of just randomly picking inputs. The first version of Adaptive Random Testing (ART) [12] was developed for numerical inputs. ARTOO introduces the notion of distance between objects and it selects objects that have the highest average distance from objects that have previously been used as test inputs and repeats this exercise in the next iteration to generate test inputs. Object distance is based on: 1. elementary distance, that is, the distance between values, which could be reference values or primitive values for primitive types; 2. type distance, which is a measure of the difference between the objects' types and 3. field distance, which uses object distance applied to the objects' fields recursively, as they themselves are objects or primitive values.

The algorithm proceeds by selecting a method r, picking a receiver object from a pool of objects. Picking or generating arguments (based on object distance) for the method r. The newly created objects are added to the pool and the method is executed with checks for any contract violations.

A study by Arcuri et al. [12] show that adaptive random testing offers minor improvements when compared to plain random testing and the benefits do not outweigh the costs incurred in object distance calculation.

Dynamic Taint Analysis

Dynamic taint analysis is commonly utilised in security research to discover information-flow vulnerabilities [164]. It executes a program and tracks which computations are affected by taint sources. Taint sources could be sources of user input, and they must be predefined for the analysis. Dynamic taint analysis can be used to enforce taint-based security policies and prevent taint-related attacks (e.g., buffer overflows, SQL injection) [36]. It can also be useful for detecting information-flow vulnerabilities by using dynamic taint analysis in conjunction with sink-aware fuzzing [160] or random testing. There are two types of dynamic taint analysis techniques for Java: systems that track only Java Strings [108] and general-purpose taint tracking systems [23]. The approaches differ in runtime overhead.

Analyses for Performance Bugs

Performance bugs is a problem related to DoS vulnerabilities. Olivo et al. [126] study redundant traversal performance bugs in Java code, limited to traversals in non-recursive functions, and a static analysis, CLARITY, to detect them.

Burnim et al. [33] present WISE, automated test generation for detecting worst-case complexity in programs. WISE uses symbolic test generation.

Jiayi et al. [190] describe Singularity, another input generation technique for detecting worst-case performance bugs in Java programs. Singularity uses a greybox fuzzing technique that looks for critical input patterns modelled as recurrent computation graphs (RCGs). Their technique reveals performance and DoS-related bugs in real world programs. Other fuzzing approaches include SlowFuzz and PerfFuzz [102, 142].

Nistpor et al. [124] propose Toddler, an example of dynamic analysis to detect performance bugs. Toddler instruments loops and read instructions, and uses the data collected using inserted code to detect similar memory-access patterns.

Padhye and Sen [134] describe Travioli, a dynamic analysis technique for detecting data-structure traversals. Like Toddler, it is also based on instrumenting code in order to harvest trace data, from which the analysis model is built. The purpose is similar to what we try to achieve in Chapter 5, however this being a dynamic model, it has different tradeoffs between precision and recall, and its quality depends on the existence of drivers (such as unit tests) that exercise a large part of the program under analysis.

SAFER [34] is a tool that detects semantic vulnerabilities in C programs that may be vulnerable to DoS attacks using malicious inputs.

Detecting Algorithmic Complexity Attacks

Wuestholz et al. [193] discuss an approach to statically detect DoS vulnerabilities in programs that use regular expressions. The analysis has multiple stages and is conceptually similar to the analysis proposed in this paper: they first build a model to detect vulnerable structures (by reasoning about the worst-case complexity of nondeterministic finite automatons), and then devise a separate (taint) analysis to establish whether a vulnerable regular expression can be matched against an input string. The tool they have developed, Rexploiter, finds 41 exploitable security vulnerabilities in Java web applications.

Staicu and Pradel [174] investigate ReDoS vulnerabilities in Node.js servers, starting with identifying the presence of critical regular expressions that may exhibit high complexity in modules, and then crafting input for servers using those modules. They found 25 previously unknown vulnerabilities and 339 web sites that are affected by at least one of those ReDoS vulnerabilities.

Holland et al. [88] propose a hybrid approach to detect algorithmic complexity

vulnerabilities. In a static pre analysis step, they use a loop call graph to detect nested loop structures that are susceptible to algorithmic complexity vulnerabilities. In a second step, they generate input and build a regression model to infer the presence of vulnerabilities. The first step is similar to the approach in Chapter 5, but uses a different model. The approach in our study is based on the presence of higher level data structures and recursive methods which then implicitly create the nested traversals.

2.5 Conclusion

In this chapter, we discussed the aspects of software security that are important to ensure safety and security in running and using software. Namely confidentiality, availability and integrity. We also looked at the Java security model and its approach to guaranteeing safety and security properties both in the language and the Java runtime environment. We also discussed two categories of vulnerabilities: DoS and remote code execution. Finally, we looked at approaches to detecting these vulnerabilities in software using program analyses techniques.

Chapter 3

Serialisation

Serialisation is the mechanism by which program state is captured to an external format for persistence of runtime data or for procedure calls across process boundaries. It involves the conversion of internal runtime representations to a binary or text-based representation and back (deserialisation). It has been described by Herlihy and Liskov [87] as a mechanism for transferring abstract data types, and the idea was extended to support reference types in the Modula programming language. The feature is supported in many object-oriented languages including Java, C#, Python and Ruby. Serialisation-based object storage and retrieval is used for data persistence, and communications over networks. In addition to the standard library routines, alternate serialisation libraries are available with support for additional features, different data formats and better performance.

The C standardisation FAQ[1] offers guidance on how expressive a serialisation implementation must be in order to transform in-memory data to a stream. The following are relevant features:

1. The format of data for the persisted stream, either binary or text based.
2. Objects are either part of an inheritance hierarchy or not (all of the same class) and if they contain references to other objects.
3. Objects contain references to other objects, but they form a tree with no cycles or multiple references to the same object.
4. Objects to be serialised contain references to other objects, and when those references form a graph with no cycles.
5. Objects to be serialised contain references to other objects, and when those pointers form a graph that might have cycles or objects with multiple references to the same object.

[1] `https://isocpp.org/faq`

3.1 Data Formats

The data format can be self-describing or it can be opaque and rely on the client program to interpret the syntax of the data. Self-describing formats are more flexible as they do not depend on a particular program to interpret the data. The format representation can either be binary or text based. The serialisation mechanism may choose a data serialisation language such as YAML [195], XML [189] or JSON [94]. These languages offer built-in support for basic types such as strings, lists or maps and additional types can be specified by using schemas.

3.2 Constructing Values or Objects

In object-oriented languages such as Java, objects are instantiated using the `new` operator followed by the constructor for the desired type, which allocates memory for the instance and returns a reference to it, in addition to invoking the constructor method for the object. Type constructors can take arguments or they can be nullary (declared without parameters). It is the constructor's role to validate the input arguments; defensively make copies for the passed references; or normalise the supplied data before utilising it.

Serialisers may handle type instantiation and initialisation differently. Some require a nullary constructor to construct the object and use setter methods to initialise the fields with data from the stream. This can be a safe method to mutate objects. Others such as Java's binary serialisation feature, 'unsafely' construct the object and use reflection or a native extra-linguistic mechanism to set the field values. This mechanism to create objects bypasses validation, and defensive checks, and risks breaking the instance's invariants, which were originally enforced by the constructor.

The methods used to obtain values from an object graph and to reconstruct the object graph from the externalised data can be classified into two major categories: beans-based serialisers and field-based serialisers [112]. Beans-based serialisers use the JavaBean API conventions [95] to get values from the fields of source objects in the object graph to be serialised, and to set the field values of deserialised objects. As per bean conventions, object properties are accessed through getter methods and mutated using setter methods. For instance, if a field has a name `foo`, the setter method is `setFoo`, the getter method is `getFoo`, except if the field is of type boolean then it is `isFoo`. The respective setter method is `setFoo`. Field-based serialisations, for instance Java native serialisation, access the actual source or target object's fields directly instead of using any of the object's API methods.

Some types (e.g. hash tables that have a memory-dependent structure) may require a custom representation for reasons such as space efficiency or representation capability.

For types that are not linear, call backs must be defined with custom deserialisation logic as serialisation cannot simply extract state and dump it to a stream. Maps are a classic example of this. Maps are often implemented as hash tables, an efficient data structure for value lookup. A hash function is used to compute a numeric hash value, which corresponds to an address location in memory that contains the associated value. When a map is serialised, an array of key-value pairs are written to the stream and when it is deserialised the internal hash table is reconstructed.

3.3 Type Safety and Encapsulation

Sometimes the type of deserialised values may not be known at compile time and during runtime the values can be cast to specific types. To support dynamic types during deserialisation some serialisation libraries support polymorphic deserialisation (i.e. can construct any subtype of the requested type). An example is the `readObject` method in Java's object input stream class, which deserialises any object that is a subtype of the `Serializable` or `Externalizable` interface. Deserialisation obtains the relevant type information from the stream, but it leaves the decision to the programmer to select the correct type.

In the case of other serialisation libraries where the deserialised type is known at compile time, deserialisation is directed by the declared type and associated deserialisation function for that type (e.g. serde library for the Rust programming language). For polymorphic deserialisation, the stream must have type discriminators / tags to direct the construction of types. Note that even with type-directed deserialisation, type information must be included in the stream to handle subtype polymorphism or union types.

3.4 Serialisation in Java

3.4.1 Serialisation Mechanisms

Serialisation was introduced to Java by Riggs et al. [156]. Serialisation is accompanied by a matching feature to read or deserialise an object graph from a stream. Serialisation has a number of use cases, most of which are in distributed computing [117]. In addition to the standard library routines, alternate serialisation libraries are also available for Java (e.g. XStream, Kyro). Riggs et al. [156] list some of the use cases for serialisation:

- Checkpointing application state, which is taking a snapshot of an application's state so that it can be restored later in time.
- Support for persistent objects (saving in-memory state of objects to disk.)

- Marshalling of objects as arguments to be passed in remote-method invocations, and to return their results.
- Serialising objects or object graphs for storage/retrieval as binary objects from databases.

In Java, serialisation is the foundation of several important platform features and protocols, including Remote Method Invocation (RMI), Java Naming and Directory Interface (JNDI), Java Management Extensions (JMX) and Java Messaging Service (JMS) [119] .

Binary Serialisation

The Java platform allows the transfer of Java bytecode as classes, which enables transmitted code to be started on a different machine. However, the transfer of state is necessary to resume a computation with the state it had prior to transmission. To facilitate this, Java offers the `ObjectOutputStream` and `ObjectInputStream` classes that provide (de)serialisation functionality. Binary serialisation involves invoking the `writeObject` method of an `ObjectOutputStream` with the target object as the argument. The target object must implement the marker interface, `Serializable`. Another condition is that the class must be able to access the first nullary constructor of its first non-serialisable superclass. Along with object data, serialisation writes meta information to identify the type of each object and the relationships between objects within the stream. In other words, a serialised object graph. By default all non-transient (fields without the `transient` modifier) and non-static fields are written to the stream as the object's data. The stream format is described in the "Java object stream serialization protocol"[2]. The format supports dynamic proxies, enums, strings, primitive data, null and object references.

Deserialising from a stream can be accomplished by calling the `readObject` method of the stream. It recreates an object graph that is isomorphic to the instance that was serialised to the stream. In addition to the default mechanism to read and write classes from and to a stream, objects may define custom serialisation and deserialisation logic by implementing the methods:

- `writeObject` for a class `T` specifies the contents that are written to the output stream when an object of type `T` is serialised. This can include custom logic and invoke `write*` methods for different types of data (strings, primitive types).
- `readObject` for a class `T` specifies the behaviour when an object of type `T` is deserialised from an input stream.

[2] https://docs.oracle.com/javase/8/docs/platform/serialization/spec/protocol.html

- `writeReplace` for a class `T`, writes the object returned by the method instead of itself to the output stream. This method is useful for securing serialisation using a seperate type as the serialised form for `T`.
- `readResolve` for a class `T` deserialises to the object returned by this method.

Custom deserialisation may be required where the in-memory state must be recomputed when the object is reloaded. This is the case for data structures like hash tables that organise objects in groups based on their hash value that might be based on memory addresses which are local to the machine on which the structure is created. These methods are automatically invoked by the serialisation routines. The `Externalizable` interface with `readExternal` and `writeExternal` is another way for developers to customise binary serialisation. These methods allow more control over what is written to and read from the stream during (de)serialisation.

Listing 3.1 shows Java binary serialisation in action. The listing demonstrates the implementation of a class with custom serialisation and a main method that proceeds to create an object of the class, write it to a stream and read it back.

```
1 public class Foo implements Serializable {
2   public Object bar;
3
4   // Hook to perform custom serialisation
5   private void writeObject(ObjectOutputStream o)
6     throws IOException {
7     o.writeObject(bar);
8
9   }
10  // Hook to perform custom deserialisation
11  private void readObject(ObjectInputStream o)
12    throws IOException, ClassNotFoundException {
13    bar = (Object) o.readObject();
14
15  }
16
17  public static void main(String[] args) {
18
19    Foo f = new Foo(); f.bar = "";
20    // serialises the Foo instance to a file
21    FileOutputStream fos = new FileOutputStream("testfile");
22    ObjectOutputStream oos = new ObjectOutputStream(fos);
23    oos.writeObject(f);
24    oos.close();
25
26    // deserialises the Foo instance that was written to the file
27    Foo obj;
28    FileInputStream fis = new FileInputStream("testfile");
```

```
29      ObjectInputStream ois = new ObjectInputStream(fis);
30      obj = (Foo) ois.readObject();
31      ois.close();
32    }
33 }
```

Listing 3.1: Java Binary serialisation

Other Serialisation Mechanisms

Default binary deserialisation[3] is not the only available technology for Java object persistence. Others include XStream, an XML-based third-party serialisation library, for which similar vulnerabilities have been reported, XML Encoder/Decoder and Kyro used in Twitter's Chill and Scalding projects. The existence of multiple technologies for serialisation suggests a larger surface for weaknesses, and hence more possible vulnerabilities of this class.

Distributed computing frameworks such as Apache Storm [117] and Apache Spark use alternative serialisation methods for efficiency reasons [117]. Amongst these alternatives are Jackson [2], Kryo [65], Protocol Buffers [78] and XStream [194]. Kryo and XStream are particularly interesting because they let the user serialise any class whereas with native binary serialisation the class must implement the `Serializable` marker interface.

3.5 Security Risks of Serialisation

Attackers who have access to an application's endpoint that unsafely utilises serialisation can craft serialised objects that violate the object's invariants or contain malicious data. Some weaknesses in Java serialisation that makes it vulnerable to attack are:

- Default deserialisation leaves objects open to manipulation of invariants and unauthorised access as deserialisation is field-based [26].
- Custom serialisation has to be implemented with defensive checks to ensure deserialised objects are valid and their properties cannot be accessed illegally [26].
- Implementing defensive deserialisation can be complex [26].
- An application that uses binary deserialisation can inadvertently instantiate any class on the class path.
- Calls the `readObject` method of an object of a type dictated by the data from the stream

[3] https://docs.oracle.com/javase/8/docs/technotes/guides/serialisation/index.html

- Java's security mechanism cannot prevent trusted code from deserialising serialised objects crafted by an attacker and this can lead to code execution without restrictions.

The improper use of Java serialisation can compromise application safety [109] which may result in attacks ranging from service unavailability or degradation to arbitrary code execution. Holzinger et al. [89] present a comprehensive study of Java vulnerabilities. The authors identify 15% of the attacks in the study as attacks related to serialisation and two DoS exploits, one caused by disk space exhaustion and the other a result of a bug in garbage collecting deeply nested structures. They present a metamodel prepared from a large body of exploits to determine the commonalities in attacks that identify Java language features and weaknesses that cause them.

Trusted code relies on the encapsulation of Java as a means to isolation, that is preventing external code from accessing private fields and methods [77]. Accesses to private fields and methods are checked at compilation and at runtime to ensure that encapsulation properties are not violated. However, language features such as subclassing and deserialisation may break this guarantee. Goichon et al. [77] cite Java access control flaws which demonstrate privileged deserialisation as an actual security threat and list CVEs for these vulnerabilities. Privileged deserialisation of untrusted data results in untrusted code being able to escalate privileges and basically instantiate any class or run any method thereby thwarting the security policy of the system.

An application that uses binary deserialisation can inadvertently instantiate any class on the class path. With the use of serialisation, fields that are otherwise inaccessible can be modified and, hence, corrupted [26]. Unchecked deserialisation of corrupt data can lead the application to an unexpected state. An attacker who has access to the communication medium can craft serialised objects that potentially break the object's invariants [26]. Custom deserialisation has to be implemented with defensive checks to ensure that deserialised objects are valid [26]. However, implementing defensive deserialisation can be a complex task as serialisation is a feature that works against the Java security model's goals [89].

3.6 Vulnerabilities

The list of CVEs show 72 Java serialisation vulnerability reports since 2003. Half of these were reported over 2015 and 2016. A significant number of them are a consequence of a single deserialisation weakness in Apache Commons Collections (ACC), and nearly all of these vulnerabilities have a critical impact as they allow arbitrary code execution with the privileges of the user running the targeted application.

In this section, we present a detailed discussion of serialisation vulnerabilities and some instances of them in libraries.

3.6.1 Injection Vulnerabilities

Custom serialisation using `readObject` or `readExternal` may have call sites with method calls on objects that are read from the stream, which can be controlled by an attacker. Further, attackers can construct object graphs that lead to sequences of method calls that may end up with attacker-controlled data flowing as arguments to sensitive methods resulting in dangerous operations. This could be shell commands to `Runtime.exec` or input to reflective method calls. Such method call chains or sequences are called *gadgets chains* (similar in concept to ROP as discussed in Chapter 2, Sec. 2.2.2). The conditions that must be satisfied to execute a gadget chain that ends in a dangerous method call are: 1) the chain must start with a class that implements custom deserialisation using `readObject` (such classes are also known as trampoline classes); 2) the object graph with the trampoline class object at the root must be constructed such that it results in the desired sequence of calls when it is deserialised; 3) the object graph must be constructed with objects belonging to classes on the class path of the target application.

Several serialisation-related *arbitrary code execution* vulnerabilities were presented by Frohoff et al. in [70]. The most well known of these vulnerabilities was the one discovered by Christopher Frohoff and Gabriel Lawrence in version 3.x of the Apache Commons Collections library. It is possible due to a gadget chain constructed using classes from the library that can result in operating system command injection. The listing in Figure 3.2 shows code for producing a payload consisting of the chain of gadgets for the ACC InvokerTransformer vulnerability. The exploit consists of an object graph constructed in such a way that it causes the side-effect of executing an arbitrary command during deserialisation.

The object graph of the payload used in the exploit has a `HashMap` at the root. The map has a single entry with a `TiedMapEntry` object as the key. The `TiedMapEntry` is a structure with a map accompanying a specified key. On deserialisation, the hash of this structure is computed to rebuild the hash map object. Computing the hash causes the retrieval of the entry with the specified key from a `LazyMap`. The `LazyMap` has an associated sequence of transformers that transform the value if an entry is not found in the underlying map. The chain consists of a series of `InvokerTransformer`s that end with invoking the `exec` method on the `Runtime` class with arguments read from the serialised data. This can potentially result in executing an arbitrary command as the data structure is rebuilt on deserialisation. The command passed to the method is executed when the payload is deserialised by a trusting application, i.e. without any of

the mitigation measures.

```
1 import org.apache.commons.collections.functors.InvokerTransformer;
2 import org.apache.commons.collections.functors.ChainedTransformer;
3 import org.apache.commons.collections.Transformer;
4 import org.apache.commons.collections.functors.ConstantTransformer;
5 import org.apache.commons.collections.keyvalue.TiedMapEntry;
6 import org.apache.commons.collections.map.LazyMap;
7 import java.util.Map;
8 import java.util.HashMap;
9
10 public class PayLoad {
11
12     public HashMap obj(String cmd) {
13
14         final String[] execArgs = new String[] {cmd};
15         TiedMapEntry tm = new TiedMapEntry(null, null);
16
17         final Transformer[] transformers = new Transformer[] {
18             new ConstantTransformer(Runtime.class),
19             new InvokerTransformer("getMethod", new Class[] {
20                     String.class, Class[].class }, new Object[] {
21                     "getRuntime", new Class[0] }),
22             new InvokerTransformer("invoke", new Class[] {
23                     Object.class, Object[].class }, new Object[] {
24                     null, new Object[0] }),
25             new InvokerTransformer("exec",
26                                    new Class[] { String.class },
    execArgs),
27             new ConstantTransformer(1) };
28
29         Map innerMap = new Flat3Map();
30         Map lazyMap = LazyMap.decorate(innerMap, new ChainedTransformer(
    transformers));
31         TiedMapEntry entry = new TiedMapEntry(lazyMap, "foo");
32         HashMap map = new HashMap();
33         List key = new ArrayList();
34         map.put(key, "foo");
35         key.add(entry);
36         return map;
37     }
38 }
```

Listing 3.2: ACC InvokerTransformer vulnerability (payload construction only)

The Apache Commons Collections serialisation vulnerability demonstrates the mechanics of how a set of gadgets can be strung together to produce unintended side-effects on deserialisation. The trampoline class in the gadget chain is `HashMap` with `hashCode`

called from `readObject`.

Serialisable `InvocationHandlers` which are a part of how Java implements dynamic proxies (interfaces to objects where messages are intercepted and dispatched to the `InvocationHandlers`) enable the construction of more gadget variants [120]. Embedded languages like Groovy, BeanShell and JavaScript add to this diversity and vulnerabilities have been discovered that involve these languages. An even more interesting variant is a vulnerabilitiy in the XSLT compiler which can be used to generate and load a class defined from data read from the serialised stream.

Serialisation vulnerability CVE-2016-2510 [53] that affected BeanShell, a scripting language for Java that encompasses an interpreter with the ability to dynamically execute Java source code, involves code injection and the use of dynamic proxies. The trigger method is `compares` in `PriorityQueue`. An invocation handler class in BeanShell, `XThis.Handler`, initialised with the injected code, is used as the handler for a proxy `Comparator` that is set as the comparator for the `PriorityQueue`. On deserialisation, `PriorityQueue`'s custom deserialisation rebuilds the queue that requires invocations to the comparator, resulting in the execution of the injected code. This particular handler delegates all intercepted method calls to the injected code.

Another library, C3P0, which enhances the capabilities of existing JDBC drivers, has a class `PoolBackedDataSourceBase` with an implemented `readObject` method, which causes the class to load a remote object and instantiate it upon deserialisation. The connection URL and class name are stored in
`ReferenceSerialized` which is a property of `PoolBackedDataSourceBase` read by the class during deserialisation. In this case, the sink is calling `newInstance` on an object acquired by loading a class via a `URLClassLoader`.

Clojure supports the dynamic creation of objects that implement one or more Java interfaces and/or extends a Java class. The created objects are anonymous classes that implement the `IProxy` interface, with a map from method names as strings to Clojure functions. Invocation of a method resolves to the corresponding target function in the map. In the case of this vulnerability, the Clojure jar file contains an ahead-of-time compiled anonymous class, that extends `AbstractTableModel`. The invocation of the `hashCode` function of this class by a deserialisation trigger such as `HashMap` can results in the execution of arbitrary Clojure code. The runtime passes the execution of the script to the Clojure function `clojure.main.eval-opt`, which reads and evaluates Clojure expressions that are in text format. This function is mapped to the `clojure.main$eval_opt.invoke` method in bytecode.

The Extensible Stylesheet Language Transformations (XSLT) compiler included with Java allows loading and instantiation of classes that are defined from deserialized data. Briefly, the design of the Java API for XML processing (JAXP) includes

XSLT processing capabilities by compiling XSL stylesheets into "translets". Compiled translets can be cached within a `Templates` object as byte arrays. Templates, which are serialisable, have a getter method, `getOutputProperties` and `newTransformer` method, both of which request the class loader to define and load the class defined in the byte array.

The Jython gadget, reported in CVE-2016-4000, also involves a serialisable invocation handler that calls injected python bytecode from the stream after intercepting any method. A similar vulnerability reported for Groovy, CVE-2015-3253, uses `ConvertedClosure` an invocation handler that intercepts any method and invokes a deserialised `MethodClosure` object from the stream, which allows an attacker to call any method on a deserialised object.

The `AnnotationInvocationHandler` in the JDK library implements handling for annotation objects. It has a class object for the annotation type and a map for properties and their corresponding values. A method call on the interface is intercepted by the handler and returns the matching value from the map. In effect, this handler can act as a function memoizer, intercepting any method and returning any value. Another way the gadget can be exploited is by calling the equals method on the proxy. This results in the handler invoking all methods on the proxied object.

The Mozilla Rhino JavaScript engine, included in the JDK, has a gadget that is reachable via calling the `toString` method on a serializable `NativeError` object. The gadget allows a `Method` to be instantiated on deserialisation from the stream, and the method to be invoked on a deserialized object.

In the Javassist/Weld gadget, any method can be executed on a stream-defined object on deserialisation. In the ROME gadget, invocation of the `hashCode` method results in a chain from `Object.toString` to calling getter methods on another object from the stream. This can be combined with the Xalan gadget resulting in arbitrary code execution.

The `SerializableTypeWrapper` gadget in the Spring framework lets attackers perform remote code execution in combination with an attack gadget. `SerializableTypeWrapper$MethodInvokeTypeProvider` permits the execution of any method in a class from the class path.

3.6.2 Expression Injection

Expression injection is a class of vulnerabilities allowing attacks by controlling input data that is evaluated by an expression language interpreter [175]. Various expression language technologies are available for the Java platform. These include MVEL, JavaServer Pages Expression Language (JSP EL), Apache Commons Object-Graph

Navigation Language (OGNL) and Spring Expression Language (SPEL). CVE-2017-5638 was reported in late 2017 for Apache Struts. The application processes an untrusted OGNL expression from the content headers of an HTTP request. Though serialisation was not involved in that particular vulnerability, a deserialisation vulnerabilities based on EL is available for Apache MyFaces, a Java EE user interface implementation.

3.6.3 System Resource Access

CVE-2016-1000031 reported for Apache Commons FileUpload library suffers from a deserialisation vulnerability in the custom deserialisation implemented in `readObject` and `writeObject` for the class `DiskFileItem`. Manipulating the `cachedContent` and `repository` fields of a serialised `DiskFileItem` object gives attackers the ability to create and copy files on the system. An identical vulnerability was reported for Apache Wicket allowing remote attackers to write to arbitrary files by inserting a NULL byte in the file name of the serialised object. A simple gadget of two objects, URL and a HashMap can be used to trigger DNS-lookups on deserialisation.

3.6.4 DoS

Serialisation-based DoS attacks can be classified into three categories: algorithmic complexity vulnerabilities, crafted object graphs and unchecked initialisation of data structures. We discuss these categories in detail below.

Algorithmic Complexity

Algorithmic complexity vulnerabilities are defects that let attackers induce worst-case performance in computing small inputs. They can be leveraged in serialisation for DoS attacks. Crosby et al. [47] report a hash collision vulnerability in Perl's implementation of associative arrays/hash tables, which resulted in inserting many keys with the same hash code into a hash table, triggering worst-case performance, $O(n)$ rather than the typical-case performance, $O(1)$. As hash tables are used to process HTTP requests this can be used as an attack vector on vulnerable hash table implementations. YAML and JSON could also be used as attack vectors. The hash collisions issue was reported for multiple programming languages by Julian Wälde and Alexander Klink [5]. In the case of Java, `String.hashCode()` hash function was not sufficiently collision resistant. As `hashCode()` is used in the implementations of `HashMap` and `Hashtable` classes it can be used in attacks. In Java version 7 and since, the HashMap implementation has been changed to use a `TreeMap` instead of a `LinkedList` to store values. In response to the discovery of the vulnerability, other languages such as Perl and Ruby have changed the hash function to SipHash [17] which mitigates hash-flooding vulnerabilities.

Crafted Object Graphs

Object graphs that are recursive or that represent nested data can also be problematic for deserialisation. Attempting to deserialise object graphs with cycles can lead to stack overflow errors. Billion laughs-style attacks such as XML Entity Expansion (XEE) attack and *SerialDoS* are enabled by hierarchical data with internal cross-references. In XML, it is the nested entities that are used for XEE attacks and in *SerialDoS*, serialised nested object graphs with internal cross-references are used as deserialisation attack vectors. A similar expansion of the structure to a string will occur when it is stringified (e.g. using the `toString()` method in Java.)

The vulnerabilities that we study are closely related to the well-known *billion laughs* attack on XML parsers [39]. It consists of an XML document with an embedded Document Type Definition (DTD) section that defines a list of nested XML entities where each entity's definition contains references to the preceding entity definition. Entity expansion on parsing results in a string, which has a length exponential in the depth of the nesting, that in effect causes the service to degrade or fail from memory exhaustion. In the listed XML code, the parser would expand to 10^9 "lol" strings taking up nearly 3 gigabytes of memory.

```
<?xml version="1.0"?>
<!DOCTYPE doc [
 <!ENTITY l "lol">
 <!ELEMENT doc (#PCDATA)>
 <!ENTITY l1 "&l;&l;&l;&l;&l;&l;&l;&l;&l;&l;">
 <!ENTITY l2 "&l1;&l1;&l1;&l1;&l1;&l1;&l1;&l1;&l1;&l1;">
 -- ENTITY l3...l8 omitted for brevity
 <!ENTITY l9 "&l8;&l8;&l8;&l8;&l8;&l8;&l8;&l8;&l8;&l8;">

]>
<doc>
&l9;
</doc>
```

Listing 3.3: XML Billion Laughs

The basic idea of billion laughs can be ported to an attack on Java programs as shown by Coekaert's SerialDoS vulnerability [37]. In SerialDoS, shown in Listing 3.4, the equivalent of the nested entity references in an XML element is a serialised collection of nested sets.

```
import java.util.*;
...
Set root = new HashSet();
Set s1 = root;
Set s2 = new HashSet();
for (int i = 0; i < depthN; i++) {
    Set child1 = new HashSet();
    Set child2 = new HashSet();
    child1.add("foo");
    s1.add(child1);
    s1.add(child2);
    s2.add(child1);
    s2.add(child2);
```

```
14     s1 = child1;
15     s2 = child2;
16 }
17
18 root.hashCode();
```

Listing 3.4: SerialDoS payload construction

The problem occurs if `root.hashCode()` is invoked. The `hashCode` of `HashSet` is recursive, i.e., it is computed using the hash values of the elements of the respective sets. This leads to a computation that is exponential in `depthN`. The first enabling property for this attack is the shape of the data structure, forming a network of cross-referencing parent-child relationships, where each child node is referenced by more than one (in this case, two) parent objects. The resulting object graph for the listing is shown in Fig. 3.1. The second ingredient is the recursive method that operates on this structure, `hashCode()` in this case.

Finally, there must be a way to parse the format to an internal representation with the same topology and trigger the traversal over it. The fact that the method can be reached from the entry point of a program can be statically determined by examining direct paths from the entry point method to the target method in the call graph.

In the case of SerialDoS, the trigger is the deserialisation API. Using any known program surface that deserialises external data, this structure can be serialised and passed as input into the program. The deserialisation of the `HashSet` instances encountered in the stream will then invoke `hashCode()` via a call graph chain from `readObject()`, and the malicious computation is activated.

Redundant traversals can be solved using dynamic programming in some cases. However, this solution is not available in the case of SerialDoS where it is not a programming defect that gives rise to the vulnerability. In SerialDoS, each method invocation happens within another context (i.e., state of the stack), and for a programmer to cache intermediate results (hashcodes for objects at each level) would require knowledge about the state of the heap and stack during each invocation.

Another issue in recursive operations over such structures is uncontrolled recursion (CWE-674 [54]) due to arbitrary nesting depth, or cycles in the structure. It can lead to infinite recursion that results in a runtime fault from stack memory exhaustion. In our earlier work [58] we studied the impact of resource (CPU, stack and memory) exhaustion / uncontrolled recursion on different Java application servers using serialisation as an attack surface. We adapt SerialDoS to construct a payload dubbed Pufferfish that leads to heap memory exhaustion. This involves each invocation allocating memory that is referred to in the calling context. An example is the stringification of nested containers in Java by invoking the `toString` method on the root of the object graph.

[4]Note that parents have two children, but children also have two parents. The graph is abstracted in the sense that intermediate references to elements in `java.util.HashSet` are not shown.

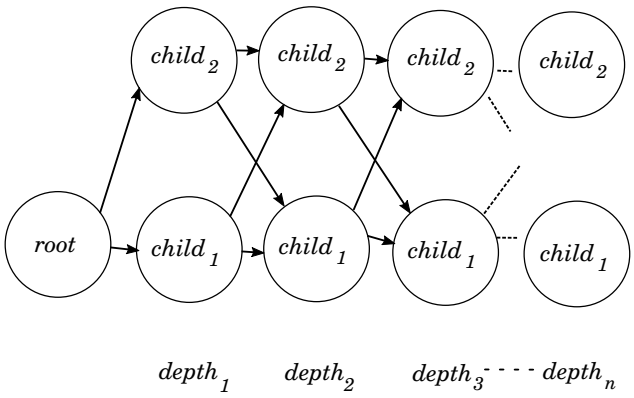


Figure 3.1: Many-to-many topology in object graphs[4]

We showed that such attacks can degrade server performance when they consume CPU resources, and memory-intensive attacks can render the service completely unresponsive.

Unchecked Initialisation

Values may be used from the stream to initialise data structures such as the size of arrays. This can allow attackers to exhaust heap memory of an application which performs unchecked deserialisation. A case is where during custom deserialisation, the code allocates memory for an array (`new T[size]`) for any type `T`, where the value for the variable `size` is obtainted from the stream using `ois.readInt()`. An example of a vulnerability is CVE-2018-2677.

3.6.5 Serialisation-Related Vulnerabilities in Other Languages

Serialisation related vulnerabilities are common in other languages, and they generally fall under the *untrusted input validation* class of vulnerabilities [187]. CVE-2013-3171, CVE-2012-0161 and CVE-2012-0160 [40] [41] [45] document arbitrary code execution using serialisation vulnerabilities in the Microsoft .NET platform. Python documentation warns against using its serialisation module, *pickles*, for deserialising untrusted data. CVE-2012-4406 [42] documents a pickling related vulnerability in a distributed object storage application written in Python. A Ruby DoS attack reported in CVE-2013-0269 [43] concerns how parsing JSON can cause memory exhaustion for maliciously crafted JSON data. During parsing data can be coerced into Ruby symbols - which are not garbage-collected, resulting in an exploitable memory leak.

3.6.6 Exploiting Deserialisation Vulnerabilities

Deserialisation vulnerabilities in libraries can be exploited when an application deserialises untrusted data. Applications can accept such data in various ways: through file uploads, HTTP requests (either cookies or GET, POST requests) and also potentially through local files residing on the same machine or network that hosts the application. [31] discuss how such exploits can occur in real world software, examples being the Jenkins and JBoss servers. In the case of Jenkins, the application had a command-line interface for remoting that used embeded serialised objects. An attacker with access to the interface can exploit vulnerable classes in the application's class path. In the other example cited in the same article, the JBoss application server had a public servlet endpoint that accepted serialised objects and once again, an attacker with access to the endpoint could instigate attacks.

A general-purpose measure against exploits based on untrusted data is to sign the input data and verify it at the point of deserialisation. However, there are downsides to this approach: signing keys can be compromised and there is the additional overhead of managing the required PKI infrastructure and the practicality of deploying it. More specific measures are available to address deserialisation vulnerabilities or attacks. These techniques and their shortcomings are discussed in Chapter 8

3.7 Recall of Static Analysis for (De)serialisation Features

Static analysis tools are generally useful for detecting taint-related vulnerabilities and are applicable for serialisation. For the PHP language a tool is available [56] that detects and automatically generates code reuse attacks (gadget chains) for deserialisation. Frohoff's [70] `InspectorGadget`, which is limited to finding if an application uses serialisation and does not devirtualize calls.

To study the recall of general-purpose static analysis tools for detecting serialisation vulnerabilities, we developed a benchmark of serialisation patterns and used it as input for a taint analysis that also utilises points-to and call graph construction. The purpose of the study was to explore potential static analysis tools and approaches to use for detecting deserialisation vulnerabilities. This study informed the design of the hybrid analysis in Chapter 7.

The benchmark from this study has multiple deserialisation patterns that are exercised by a driver program (in Listing 3.6). The call to `readObject`, which returns the deserialised object, is set as the source of taint.
We also set `java.io.ObjectInputStream::readString(boolean)` as an additional source, which is the method that instantiates strings read from the binary stream during deserialisation. We use `PrintWriter::write` as a sink. We evaluated FlowDroid

and Doop with this benchmark. The objective of the evaluation was to study the recall of the tools under the various scenarios.

3.7.1 Micro-benchmark of (De)serialisation Features

The benchmark contains five patterns as listed in Table 3.1 and shown in the code Listings 3.5 and 3.6. The first pattern, *devirt-taint*, is used to check for tainted information flow that involves points-to computation and call graph reachability. (De-)serialisation offers an extra-linguistic mechanism to construct objects, avoiding constructors that can result in unsound call graphs (if it uses more precise call graph algorithms that rely on runtime types of variables for devirtualising target methods of method calls) and imprecise points-to results. The pattern has code that deserialises data from a stream and constructs an object, and then invokes a method on this object. The client class is not aware of the actual type of the receiver object, as the code contains no allocation site. This pattern is also in our micro-benchmark of dynamic language features in Java programs. An additional part of the pattern is that it also involves tainted information flow to the prescribed sink. This is the only pattern that involves devirtualization. The other four patterns are used to check if various different types of flow can be detected. The *field-taint* pattern involves a flow from a deserialised object's field to the sink. The *custom-taint* pattern has a tainted flow that is triggered within the custom deserialisation code (`ObjCustom` in Listing 3.5) for an object. The *container-taint* pattern has a value flowing from a container field of the deserialised object to the sink. The *basic-taint* pattern is the simple case where the deserialised object is a string.

Table 3.1: Taint flow patterns in micro-benchmark

Pattern	**Description**
devirt-taint	Tainted value from object field flows to sink (devirtualization required)
field-taint	Tainted value from field flows to sink (no devirtualization)
custom-taint	Tainted value from field flows to sink (custom deserialisation)
container-taint	Tainted value from container flows to sink
basic-taint	Tainted value flows to sink

```
1
2 public class ObjContainer implements Serializable {
3     public List<String> stsElements;
4 }
5
6 public class ObjCustom implements Serializable {
7
8     public String field;
9     public List<String> fields;
10    public transient String transientStr;
11
12    private void readObject(java.io.ObjectInputStream s)
```

```
13              throws IOException, ClassNotFoundException {
14          s.defaultReadObject();
15          transientStr = field;
16          PrintWriter writer = new PrintWriter(new FileWriter("sink"));
17          writer.write(this.field); // bad
18          writer.write("abc"); // ok
19          writer.write(fields.get(0)); // bad
20    }
21
22    private void writeObject(java.io.ObjectOutputStream s) throws IOException {
23        s.defaultWriteObject();
24    }
25 }
26
27    public class ObjStrFld implements Serializable {
28      public String strField;
29
30      public int hashCode() {
31          PrintWriter writer = null;
32          try {
33              writer = new PrintWriter(new FileWriter("sink"));
34          } catch (IOException e) {
35              e.printStackTrace();
36          }
37          writer.write(strField);
38          return strField.hashCode();
39      }
40 }
```

Listing 3.5: Serialisable Objects

```
1
2 public class Deserialisation implements Serializable {
3   public  static void main(String[] args)  throws Exception {
4     FileInputStream fis = new FileInputStream(new File("file"));
5     ObjectInputStream oos = new ObjectInputStream(fis);
6     PrintWriter writer = new PrintWriter(new FileWriter("sink"));
7
8     Object obj = oos.readObject();
9
10    obj.hashCode();
11
12
13    if (obj instanceof ObjCustom) {
14      writer.write(((ObjCustom)a).transientStr);
15    }
16    if (obj instanceof ObjContainer) {
17      writer.write(((ObjContainer)a).stsElements.get(0));
18    }
19    if (obj instanceof ObjStrFld) {
20      writer.write(((ObjStrFld)a).strField);
21    }
22  }
23 }
```

Listing 3.6: Deserialisation Driver

3.7.2 Evaluation

The experiments were performed on an Intel(R) Core(TM) i7-8700 CPU @ 3.20GHz with 64GB of RAM on Linux Ubuntu 18.04.3. DOOP[5] was run using the Java 8 platform as implemented in Oracle's version 8 of the JDK (build 1.8.0_221-b11).

3.7.3 Results

The recall results for the patterns evaluated for FlowDroid and DOOP are shown in Table 3.2.

Table 3.2: Micro-benchmark results.

Pattern	**FlowDroid**	**Doop**
devirt-taint	NO	YES
field-taint	YES	YES
custom-taint	NO	NO
container-taint	NO	NO
basic-taint	YES	YES

For FlowDroid, the analysis was only able to detect two out of five patterns. For the *custom-taint* pattern we also checked the analysis with the custom `readObject` as the entry point for the program. Different call graph algorithms were also used to confirm if recall improved (i.e., CHA, RTA and VTA). In the case of DOOP, the analysis detected three from the five patterns.

3.8 Conclusion

We have presented an overview of serialisation, more specifically in Java and its pertinent security aspects. We then discussed the types of vulnerabilities due to deserialisation. Finally, we evaluated two static analysis tools, which track tainted information flow, on a micro-benchmark of deserialisation features to identify recall issues with respect to the use of those features.

[5]Doop version built from commit: `https://bitbucket.org/yanniss/doop/commits/ba731a63c90e94f9e94afca39ff15c5082bf868d`

Chapter 4

Research Methodology

This chapter introduces the methodology that was adopted for the research carried out in the four studies that are reported in Chapters 5-8. The studies described in detail in the respective chapters are:

- Static analysis for detecting DoS vulnerabilities in Chapter 5.
- Serialisation vulnerabilities in non-Java languages in Chapter 6.
- Hybrid analysis for detecting injection vulnerabilities in Chapter 7.
- Mitigation measures against serialisation attacks in Chapter 8.

We situate the approach within the various methodologies available for software engineering research, and justify its selection. We then describe the experiment design, systems selection and threats to validity.

4.1 Introduction

Improvements in the state of the art and effective solutions to software engineering problems (in our case, software security in the scope of serialisation) are possible through the methodological evaluation of new methods, techniques, tools and proposals in comparison to existing ones [192]. Research methods in software engineering provide guidance on how to do this. Four research methods are used in software engineering: scientific, engineering, empirical and analytical. They fall broadly under the analytic or experimental paradigms. The analytical method proposes a formal theory, which is compared to empirical observations. The scientific method involves building a model based on an observed phenomenon. The engineering method studies current solutions to a problem and proposes an improved alternative or a new one where none exists. The empirical method proposes a model, which is then evaluated using empirical studies such as case

studies or experiments. The engineering and empirical methods are variations of the scientific method where existing solutions to a particular problem are studied, changes proposed and the resulting implementation is evaluated against those solutions.

This research involves developing analysis tools to either prevent or detect serialisation vulnerabilities, and an assessment of DoS vulnerabilities for programming languages other than Java. As these projects are developmental in nature, to gain an understanding of how and why the developed tools will be useful, we measure their properties and compare them to alternatives where they are available. The engineering method is best suited for this purpose.

The strategy we followed in the research is composed of the following steps: problem formulation, study of the state of the art, designing and developing a candidate solution and finally validation through experiments. In the problem formulation step, we identify the problem and specify a formal specification for it. This also includes a literature review to identify and assess state of the art solutions in the problem domain. In designing and developing a candidate solution, we develop tools from scratch or customise available frameworks or tools.

The experimental paradigm also involves designing and executing an experiment, to support or refute a hypothesis concerning a cause-effect relationship of interest. The experiment studies the effect of changing independent variables (inputs or causes) on the dependent variables (outcomes or effect). This requires selection of instances, executing the experiment procedure, data collection and observation of outcomes, validation and reporting the results [21]. In order for an experiment to be reliable and produce scientific value, it must be designed and conducted in a principled way. The experimental design we use is described by Wohlin [192]. We use the guidelines from Jedlitschka et al. [92] for reporting experiments in this work. Adopting a standard approach for reporting the experiments makes the information accessible, clarifies how the experiment was conducted and is helpful for assessing the results and their validity. In the following sections, we discuss experiment design and how we selected the systems for the studies, and threats to validity.

4.2 Threat Model

We assume that the attacker has access to an endpoint that accepts serialised or formatted data. In Chapter 8, we assume the availability of any sanitisation features (e.g., safe loading for YAML). We also assume that the adversary is able to craft input that have traversal vulnerabilities using known patterns or other techniques to trigger worst-case execution. The adversary is able to exploit applications or services that parse input formats to carry out DoS attacks (e.g. configuration files, upload files to instigate second-order attacks). These assumptions are valid as a large number

of applications use YAML for DSLs/configuration (e.g. Swagger, Kubernetes, build pipelines). We also assume the adversary may be able to locally or remotely exploit object deserialisation with knowledge of classes on the class path.

4.3 Experimental Design

In this section, we describe the general approach to experiment design for the listed studies. The general theme of research design in Chapters 5 and 7 (static analysis for DoS vulnerabilities, hybrid analysis) is to develop security analyses to detect serialisation vulnerabilities or attacks, and to evaluate their efficiency and effectiveness against selected instances of programs or libraries (that contain either labelled, non-labelled vulnerabilities). Where alternate program analysis tools are available, a comparison is also part of the research methodology. In Chapter 6 (non-Java languages), a specific set of libraries are selected based on language and usage popularity. The specific parts of the methodology for a respective experiment are described in detail in the relevant chapters.

The stages of the experiment are:

- **Scoping**: Analyse analysis tool for the purpose of evaluation with respect to efficiency and effectiveness from the point of view of the security analyst in the context of running the analysis on a selected set of instances of programs (Chapters 5 and 7). Analyse libraries for occurrence of a defined class of vulnerabilities for the purpose systematically identifying prevalence of these vulnerabilities in the context of a selected dataset that represent various programming languages (Chapter 6).

- **Planning**: Design the experiment, which is determining independent and dependent variables and the values for the independent variable (also referred to as treatments). Threats to validity are assessed and mitigations put in place.

- **Operation**: This stage consists of preparation (e.g. computing resources and the subjects), execution of the experiment as per the plan and validation, which is the verification of the data collected from execution.

- **Analysis and interpretation**: Analyse and evaluate the collected measurements.

- **Presentation and package**: Present the research and related experiment with summarised results.

4.4 Systems Selection

To address external validity threats, which are discussed in the next section, the instances used in the experiments are clearly defined in the respective chapters. For the first study the dataset used in the experiment is an industry-accepted collection of Java libraries with vulnerabilities. The second study is a set of libraries that are external format parsing libraries that are popular in the Java ecosystem. The final study uses libraries for the specific format that we studied, which was again based on the popularity of the library for the programming language. Language and library popularity are based on usage in the industry (measures for which are available from library repositories or sites that collect metadata from those repositories, such as usage statistics from Maven and number of forks on Github).

4.5 Threats to Validity

Validity threats are possible risks in the design and execution of empirical studies. They limit the reliability of the studies and the extent to which the results can be generalised to instances in the real world that are outside experimental settings. A framework for validity threats is available in Wohlin [192], which categorises them as: internal, external, construct and conclusion validity. We identify specific risks within these categories, and how we address them in our experiments.

Internal validity holds if there is a causal relationship between a treatment and the outcome in an experiment, and it is not the result of a factor that researcher has no control over.

To ensure internal validity, we have made the parameters of the experiment clear. Such as algorithmic configurations of the tools/platforms we used. Data collection and execution are systematic and have been described as such. The instances of libraries are not synthetic but ones that are in actual use by the community.

External validity is concerned with generalisation of observed results to larger population.

We have clearly defined the instances of libraries used in the experiments, and also clarified the strategy for how they were selected and why they were used.

Construct validity covers the correspondence between theory or the hypothesis, and observation in the treatment and the outcome of an experiment. For the experiment to be valid, the treatment must reflect the cause construct and outcome must reflect the effect construct.

A risk is if a cost measure such as execution time (observed experiment outcome) reflects actual performance (effect) of the analysis. Gauging effectiveness measures such as precision is also a challenge as one of the datasets is not labelled. Manual

classification, explained in detail in the relevant chapters, was carried out and this remains a threat to validity.

Conclusion validity is about the relationship between the treatment and outcome in an experiment. It must be demonstrated that there is a statistically significant relationship.

One of the risks is that an observed outcome could be a random occurrence. This is especially a concern as there are stochastic elements to the hybrid analysis in Chapter 7. To account for random variation, we run the experiment multiple times.

Another risk is the lack of a meaningful baseline for comparison in the evaluations. The experiment to evaluate the static analysis for DoS has no baseline. In the hybrid analysis, random search is used as a baseline.

4.6 Summary

In this chapter we touched on the importance of methodologies in software engineering research before giving an overview of experimental design for the overall work. We have also listed the systems used in the experiments, and the various threats to validity and how we address them.

Chapter 5

Static Analysis for DoS Vulnerabilities

As discussed in Section 3.6.4 of Chapter 3, complexity-related DoS vulnerabilities can be due to programs recursively traversing composite data structures. Tree and graph-like data structures that are composed of parts are common in programs, and when such structures are defined recursively, it is practical to define recursive operations over them. Such an operation can potentially run in exponential time or / and space if redundant traversals are not (or cannot be) controlled. These performance-related vulnerabilities can be exploited to carry out DoS attacks on systems. Serialisation and external format parsers that utilise these data structures present opportunities for DoS attacks based on these vulnerabilities [58].

Static and dynamic analysis techniques have been used for detecting performance-related bugs in programs. However, most existing approaches for detecting them are domain-specific - for instance, detection techniques for regular expression engines [193]. Fuzzing has been used to detect performance defects in programs. However, fuzzers are inefficient by nature with normal turnaround times of hours or more [100]. Constraint-based techniques such as symbolic execution are limited when it comes to producing complex inputs and there has been limited work on using them to detect performance bugs, especially for complex inputs [125, 137].

This chapter presents a characterisation of a class of vulnerabilities, along with a novel approach to detect them, and the results of the study. An overview of the approach is shown in Figure 5.1. It is based on modelling the vulnerabilities, implementing the analysis for the Java language and constructing payloads to verify the analysis reports. Finally, constructed payloads are used to check if libraries and applications written in other languages are vulnerable as well.

We characterise the program structure that facilitates such an attack. This is broken down into three parts (we refer to these parts as the three T's):

1. **Topology:** a data structure that has a topology which allows the redundant execution of recursive code/methods.

2. **Traversal:** the presence of recursive methods that operate on the elements in the data structures identified in step one.

3. **Trigger:** an execution path, in a program, from an entry point method for the program, typically a method that loads and evaluates data, to a recursive method identified in step two.

We implemented the analysis for Java and then evaluated it on a set of 16 Java parser libraries for different data formats. The scope and impact of this study goes beyond the vulnerabilities found in the Java libraries as these libraries are used in numerous applications. We validated the vulnerabilities by constructing malicious inputs, and it turns out that some of these reveal vulnerabilities in libraries and applications written in other languages, such as Rust and PostScript.

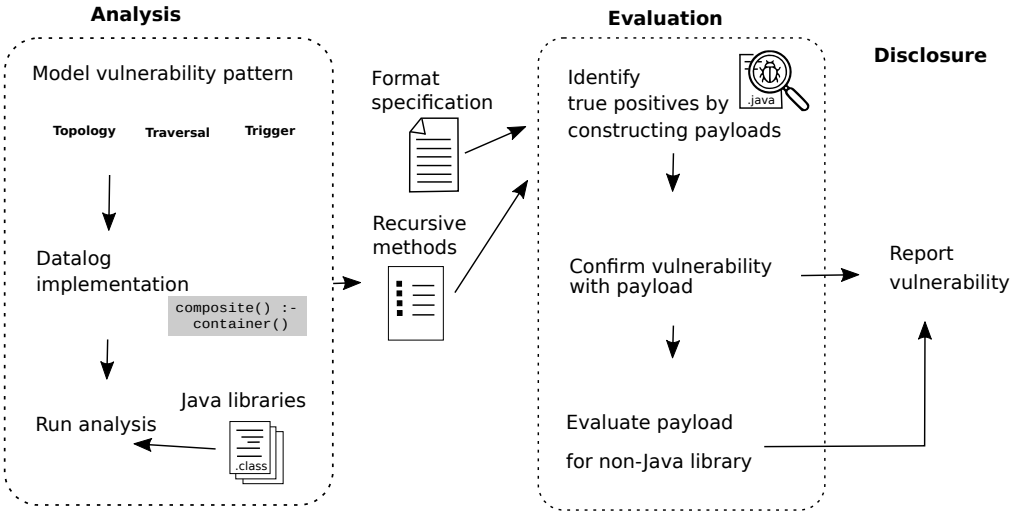


Figure 5.1: Overview of approach

In this study, we found a total of 11 vulnerabilities: Four new vulnerabilities in Java libraries using the analysis, (i.e. Apache PDFBox, PDFxStream, Apache Batik and SnakeYAML), and seven vulnerabilities in non-Java libraries found during the evaluation. All 11 issues were reported to the vendors (7 were accepted as security bugs) and four CVEs were obtained as a result.

5.1 Characteristics of the Vulnerability

In this section, we discuss the three defining characteristics, *TTT*, needed to craft DoS attacks based on vulnerabilities in code recursively traversing data structures: *topologies*, *traversals* and *triggers*.

5.1.1 Program Representation

In programming languages that support object composition for representing complex data types, and subtype polymorphism, data types consists of classes, primitive types and arrays of those types. For instance, in Java, a program consists of a set of classes with a main class as the entry point. A pair of classes in a Java program can be in a direct subtype relationship, where one class is a subclass of another. A class declares a set of methods and a set of fields, with their names and type signatures. A field, in UML parlance, defines an association relationship with the class in its type signature. If a method invocation is virtual, the target method for the call is determined dynamically during runtime. The target method of an invocation depends on the method-binding for the invocation statement. The method-binding is determined by the class of the receiver of the invocation. For instance, the method-binding for the invocation $b.m()$ can be determined by the type of the object instance that b references, the receiver of the invocation. An instantiation of a class is an object. This, itself, can be a graph of objects if we consider the objects that its fields may reference: objects as vertices and field relationships as edges.

5.1.2 Topologies

In Java-like languages, an instance of a data structure forms a graph of objects. An object, *parent*, references $child_1$ if $child_1$ is the value of a field of *parent*. Often, references are indirect via intermediate objects, in particular arrays or collections. Given an object graph, we are particularly interested in subgraphs formed by objects of some type T, where these objects have more than one predecessor and/or successor of type T. These structures can be described as following a many-to-many pattern[1]. Examples of this are common in the Java collection library, where, for instance, lists can be elements of multiple other lists.

A pattern that is very widely used, and conceptually similar to many-to-many, is *composite* [71]. A composite has containers with elements that are either containers as well, or leaf nodes. Usually, there are dedicated container and child types subtyping a

[1]Many-to-many relationships, in the database community, describe one type of cardinality of relationships between two entities.

more abstract component type, but variations of this pattern are common[2].

Whether a composite can form a many-to-many graph depends on how the parent-child relationships are represented and controlled. Often it is an implicit precondition for an operation with a composite as an input, that at most, the composite has a single reference to any of its children. This is often enforced by an explicit parent reference in the children, and an API that maintains the consistency of the parent-child and child-parent references[3].

5.1.3 Traversals

A many-to-many pattern in an object graph describes the data structure that can be exploited in SerialDoS-style attacks (discussed earlier in section 3.6.4). To actually launch an attack, a function that operates recursively on this structure must be present. Regarding these functions, or methods as they are called in object-oriented languages, there are two aspects to consider. Firstly, recursion can be simply described by the presence of strongly-connected components in the call graph. Secondly, the recursive method needs to traverse the many-to-many object graph.

Consider a method m with formal parameters $m.this, m.arg_1, m.arg_n$. Then, there must be a call graph chain linking m to itself (an invocation of the method m). Additionally, at this invocation there is one parameter that points to a child of an object that the respective parameter points to at the previous invocation.

This captures several common *traversal patterns* found in real world programs. The most obvious one is direct recursion, where the many-to-many elements are always referenced by `this`. Listing 5.1 shows an example of a simple recursion, used to count elements. Note how, at the call site in line 12, the first parameter for the called method, `this`, is bound to the child, `this.elements[i]`.

```
1  public abstract class Element {
2     public abstract int count();
3  }
4  public class Leaf extends Element {
5     public int count() {return 1;}
6  }
7  public class Composite extends Element {
8     private Element[] elements = ...;
9     public int count() {
10       int c = 0;
11       for (int i=0;i<elements.length;i++) {
12          c = c + this.elements[i].count();
13       }
14       return c;
15    }
```

[2]For instance, in `java.io.File`, state is used to determine whether an element is a container (a directory) or a child (a file). A case of a composite that uses structural instead of nominal typing.

[3]A good example for this is how the parent reference is maintained in the `java.awt.Container.add*` methods which add a child component to the visual component hierarchy.

```
16 }
```

Listing 5.1: Direct recursion over a composite

There are more complicated traversal patterns though. For instance, it is possible to implement traversals using static methods in some class, where the traversed data structure is only passed as a parameter. This can be used to program traversals implemented outside the data structure being traversed. A standardised way to achieve this is using the *visitor* pattern [71], which factors out the operation to be performed on a data structure from its implementation. Here, the traversal uses two methods: an `accept` method defined by the visited structure, and a `visit` method implemented in an external visitor that acts upon the traversed objects.

Note that the component of traversals we have discussed so far concerns traversing the graph depth-wise. Another aspect is how a method traverses the children of each parent. In the listed example, this is accomplished through the for-loop which iterates over the children in `elements`. A generalised technique is to use the iterator pattern [71], which abstracts the traversal of different implementations of collection types.

5.1.4 Triggers

The presence of the topology and the traversals itself is not sufficient to launch an attack. It is also necessary to create objects that instantiate the many-to-many pattern, and to trigger the traversal. This requires an external data structure to be translated to a vulnerable internal representation of the object graph in a data format, interpreted or otherwise processed, and in this process the traversal is activated. The activation refers to an invocation chain, from a method that is called by the library or the framework when the data is processed, to the actual traversal method. Again, SerialDoS (see Section 3.6.4) serves as a good example. It can use serialisation as a trigger mechanism. When an application deserialises an object graph, type-specific methods such as `readObject(..)` are invoked. In the case of `HashSet`, this invokes `hashCode()` on the elements, which will trigger the traversal.

In general, a trigger is a method in the public API of a library that is invoked when data is processed (either by a client program, or by the library itself), and that leads to the invocation of the traversal (method).

5.2 Modelling the Analysis

In this section, we describe how we model the topology, traversal and trigger patterns for the analysis. We formalise the analysis in graph-theoretic terms using three graphs that model different aspects of program behaviour. These definitions are language-agnostic,

and are applicable to languages with common object-oriented language features. We occasionally refer to Java for illustrative purposes.

5.2.1 Preliminaries

Type Graph

A type graph models classes and their relationships in a program written in an object-oriented (OO) language. In an OO-program, a type can either be a class, a primitive (boolean, integer, float, etc.) or an array with its component either an array, a primitive or a class. There are two kinds of relationships. First, direct subtype relationships between pairs of classes. Second, those between container classes and the type of their components. We omit primitives from the type graph as they do not have any outgoing edges. The formal definition of a type graph is as follows:

Definition 5.2.1 (Type Graph)**.** The **type graph** of a program is a directed labelled graph $\langle T, E \rangle$ where

- T is a set of vertices representing classes and arrays of classes that occur in the program, we represent vertices using their class name, with the suffix [] for array types as usual. We identify those names with the respective vertices from now on.
- $E \subseteq T \times T$ are the graph edges. We consider edges of two kinds, i.e., $E = E_{subtype} \cup E_{assoc}$, representing subtype and association relationships, respectively. Subtype relationships are the reflexive, transitive closure of the direct subtype relation.

Subtype edges between vertices represent subtype relationships. The particular rules differ between languages, for Java, they are defined in the Java language specification [79, sect 4.10]. We use the simplified notion "`B` is subtype of `A`" to say that there is a path consisting of subtype edges linking `A` to `B`. That is, the edge $(A \to B) \in E_{subtype}$. If $C \in T$ is either representing an array, or a class that is a subtype of a container type (such as Java's `java.util.Collection`) , then we call C a *container type*. If a container type is an array `A[]` then `A` is called the *component type* of the container. If the language supports the declaration of component types for container types, using some mechanism like Java's generics [79, sect. 8.1.2], `C<A>`, then `A` is called the *component type* of `C`.

Association edges represent the relationships between container types and their respective component types. Note that we only model one-to-many relationships here.

Points-To Graph

As discussed in Section 2.4.1 of Chapter 2 a points-to graph models memory during program executions. For this analysis we use a field-sensitive points-to analysis as described by the following definition of the points-to graph. Object abstractions (i.e., object allocation statements in the program) and program variables are represented by vertices. Assignment statements, field or array stores and loads are represented by edges. Such statements define the flow of values in the program being represented (flows through program variables or heap memory).

Definition 5.2.2 (Points-to Graph)**.** The points-to graph of a program is a directed labelled bipartite graph $\langle O, V, E_{alloc}, E_{assign}, E_{load}, E_{store} \rangle$ where

- V is the set of vertices representing the variables in the program
- O is the set vertices representing object allocation sites in the program
- $E_{alloc} \subseteq O \times V$ is a set of allocation edges, modelling memory allocation
- $E_{assign} \subseteq V \times V$, a set of assignment edges modelling variable assignment
- $E_{load} \subseteq V \times V$, a set of field load edges modelling field loads, labelled with the respective field name
- $E_{store} \subseteq V \times V$, a set of field store edges modelling field stores, labelled with the respective field name

Array access can be modelled similarly to field access. An array can be viewed as an object with its indices as fields.

Given a points-to graph, the objective of a points-to analysis is to infer additional *flowsTo* edges $E_{flowsTo} \subseteq O \times V$ describing the relationship between abstract values and variables pointing to them [4]. This is a computationally complex problem, usually solved by computing context-free language (CFL) reachability via a fixpoint algorithm [173, 57].

One of the main uses of the points-to graph is alias analysis. Two variable vertices v_1, v_2 *alias* if there is an object vertex $o \in O$ and paths consisting of $flowsTo$ edges from o to both variable vertices v_1 and v_2. Aliasing means that both variables can point to the same memory location. Furthermore, this can be used to define a *heap access path* between variables. A heap access path between v_1 and v_2 consists of a sequence of load edges where the destination (sink) of an edge in the sequence aliases with the source of the next edge, v_1 is the source of the first and v_2 the destination of the last edge in the sequence. This models (nested) field access in a programming language, i.e.

[4] Sometimes, the reverse *points-to* edges are inferred.

statements like `foo.f.g`. By accounting for aliases, this also covers field access with intermediate variables, in programs like `x = foo.f; x.g;`.

Call Graph

A call graph statically models the interprocedural calling (invocation) behaviour of a program. Methods are represented by vertices and interprocedural calls by edges.

Definition 5.2.3 (Call graph)**.** The call graph of a program is a directed graph $\langle M, I \rangle$ where

- M is the set of methods in the program
- $I \subseteq M \times M$ is a set of edges, $(m \to n) \in I$ means that method m has a call site with an invocation of n.

A call graph can be constructed by analysing invocation instructions found in program code. To model runtime behaviour in languages with dynamic dispatch, additional edges must be inferred (devirtualisation). There are various algorithms available for this purpose differing in their precision and efficiency (e.g. CHA, VTA)[179] as discussed in section 2.4.1. As our model also uses points-to information it is convenient to use on-the-fly call graph construction which computes both simultaneously.

Cross-Referencing Graphs

The three models defined above are widely used in static program analysis. In practice, they are often combined as on-the-fly call graph construction requires information about the type hierarchy. There are certain relationships between these models we will exploit in the analysis.

Firstly, for a given method m, the variables in the points-to graph include the return value and the parameters of this method. We denote the parameters, including the receiver of an invocation (in Java, the `this` reference), as $param(m) \subseteq V$. Secondly, call graph vertices can be associated with type graph vertices via the types that define those methods. We refer to those types as the owner of a method m, $owner(m) \in T$. Finally, allocation vertices $o \in O$ in the points-to graph can be associated with the types they instantiate, $type(o) \in T$.

5.2.2 Analysis Specification

Topologies

Given a type graph $\langle T, E \rangle$, we describe an instance of *composite* as a mapping between the two roles in the composite design pattern [71] and actual types that occur in the

program[5].

Definition 5.2.4 (Topology)**.** Given a type graph $TG = \langle T, E \rangle$, a composite is a mapping $\{cont, comp\} \rightarrow TG$ such that the following two conditions are satisfied:

- $(composite(cont), composite(comp)) \in E_{assoc}$
- $(composite(comp), composite(cont)) \in E_{subtype}$

Traversals

We define a traversal as the presence of a recursive invocation in the call graph with a heap access path from a field of a composite to an argument of the recursive call.

Definition 5.2.5 (Traversal)**.** Given the type graph $TG = \langle T, E \rangle$, the call graph $CG = \langle M, I \rangle$ and the points-to graph $PG = \langle V, O, E_{alloc}, E_{assign}, E_{load}, E_{store} \rangle$, a traversal is a configuration of vertices and edges in CG and PG such that:

- $m \in M$ is recursive in CG, i.e. CG contains a path connecting m to itself
- there is a composite c and $c(comp) \in param(m)$
- there is a heap access path in PG from that parameter $v \in param(m)$ to itself

Triggers

We define a trigger as the presence of a method that instantiates a *composite* through a chain of method invocations. Once the composite is instantiated it would also trigger the recursive traversal with the *composite* as an argument.

Definition 5.2.6 (Trigger)**.** Given a call graph $CG = \langle M, I \rangle$, a trigger is a method $trigger \in M$ in the call graph that is reachable from a program entry point $ep \in M$. There must also be a path in the CG from the $trigger$ to a $traversal$ method with the instantiated *composite* as an argument.

5.3 Methodology

5.3.1 Approach

The approach focusses on first running the static analysis to detect the TTT patterns in Java parser libraries, then using the analysis results to inform the construction of payloads to exploit vulnerabilities, and finally using the payloads for non-Java parsers

[5]The *cont* role corresponds to the *Container* role in the design pattern, whereas the *comp* roles corresponds to the *Component* role. We do not consider a particular leaf type.

(e.g., C, Python, Rust, etc.) to detect if these parsers are also vulnerable. The payloads were constructed manually using format specifications and source code indicated by analysis results. The effort required to construct payloads varied across different libraries, and this process is discussed in more detail in section 5.3.5.

5.3.2 Implementation

The analysis was implemented as an extension of Doop. Doop was selected for its expressiveness and its support for implementing custom declarative analyses. The underlying analyses in Doop compute points-to information and call graphs from an input program. We chose the Soufflé Datalog engine [13] for compiling the Datalog program, due to its performance [11].

Datalog programs are a natural way to express the graph-based algorithms [196] used in the specification of the analysis. A Datalog program consists of recursive relations expressed as rules and input facts. The computation is the monotonic logical inference of these rules until a fixpoint is reached. For instance, a datalog rule: $C(z, x) :- A(x, y), B(y, z).$ means that if $A(x, y)$ and $B(y, z)$ are both true, then $C(z, x)$ can be inferred as an additional fact. Datalog-based formulations of static analyses have been used successfully in bug and vulnerability detection [81, 107, 162].

In the Doop formulation, a rule, $A(v_1 : \mathbb{V}, ..., v_n : \mathbb{V})$ has variables which take values from a particular domain, which consist of variables, $\mathbb{V}$; object values (a value in the program is an abstraction that represents an object allocation site, e.g. `a = new Object()`) $\mathbb{O}$; method invocations, $\mathbb{I}$; types, $\mathbb{T}$; methods, $\mathbb{M}$; and numbers, $\mathbb{N}$. The primary result of the core analysis is a conservative approximation of the program's call graph and the values that a program variable may point to. We extended this with custom rules that were specified in the analysis.

Doop extracts facts from the input program using Soot [188]. The fact extraction does not include type parameters in field declarations, which are required to establish subtype relationships. We performed a bytecode pre-analysis to extract these additional facts, which are then imported during the execution of the analysis.

The datalog code for the analysis is in Listings 5.2- 5.4. Listing 5.2 defines composites that use the definitions for one-to-many types from Listing 5.3. Listing 5.4 defines rules for traversals and triggers. These rules refer to facts in the relations `FieldParamType`, `FieldName`, `FieldExtractedType`, and `Reachable`.

`FieldParamType(type, field, type_p)`, specifies that `field`, in the declared type, `type`, has a type parameter of type `type_p`. This models fields that are of a generic type with type arguments. For example, if a field is declared as `List<T> lst`, `T` is the type argument for the `List` type for the field named `lst`. `FieldName(signature, field_name)` specifies the string, `field_name` extracted from the signature. In the rule

`FieldExtractedType(type, signature)`, if a field with `signature` is an array type, `type` identifies its component type. If there is a path in the points-to-graph from variables `v_1` to `v_2` this fact is in `HeapFlow(v_1, v_2)`. In the rule, `Reachable(m)`, the method `m` is reachable from the `EntryPoint` for the program.

The datalog for modelling the composite pattern is captured in the relation, $Composite-(container : \mathbb{T}, component : \mathbb{T})$ in Listing 5.2, the relation has two variables that both belong to the domain of types. This rule corresponds to the definition in section 5.2.2.

```
Composite(container, component) :-
  OneToManyType(container, component),
  SubtypeOf(container, component).
```

Listing 5.2: Composite rule in Datalog

```
OneToManyType(type, type_leaf) :-
 FieldDeclaringType(signature, type),
 FieldExtractedType(type_leaf, signature).

OneToManyType(type, type_leaf) :-
 ArrayType(type_field),
 FieldType(signature, type_field),
 FieldDeclaringType(signature, type),
 FieldParamType(type_field, field_name, type_leaf),
 FieldName(signature, field_name).

OneToManyType(type, type_leaf) :-
 ContainerType(type_field),
 FieldType(signature, type_field),
 FieldDeclaringType(signature, type),
 FieldParamType(type_field, field_name, type_leaf),
 FieldName(signature, field_name).
```

Listing 5.3: Rules for one-to-many types

The `Composite` rule uses the `OneToManyType` rule that defines a type which can contain types that are subtypes of itself (first rule in Listing 5.3). Containment, which is defined by the two other `OneToManyType` rules, is a type having an array type field or container type field.

```
Flow(from, to) :-
  Assign(from, to);
  ReturnAssign(from, to);
  ArgAssign(from, to).

Flow(from, to) :-
  LoadField(from, to, field);
  LoadArray(from, to, index).

Flow(from, to) :-
  MethodInvocationBase(invocation, from),
  CallGraphEdge(invocation, method),
  ThisVar(method, to).

HeapFlow(from, to) :-
```

```
  HeapFlow(f, to),
  Flow(from, f).

DirectRecursiveThis(container, method_from, invocation, base) :-
  Composite(container, component),
  MethodDeclaringType(method_from, container),
  Invocation(invocation, method_from, method_to),
  Target(method_to, method_from),
  InvocationBase(invocation, base),
  ThisVariable(method_from, variable_this),
  HeapFlow(variable_this, base).

DirectRecursiveParameter}(actual, type_parameter, method, invocation) :-
  Invocation(invocation, method, method),
  FormalParam(i, method, parameter),
  VariableType(parameter, type_parameter),
  ActualParam(i, invocation, actual),
  Composite(container, component),
  SubtypeOf(container, type_parameter),
  HeapFlow(parameter, actual).

DirectThisTriggered(invocation, base) :-
  DirectRecursiveThis(container, m_from, invocation),
  Reachable(m_from),
  ValueType(value, container),
  VariablePointsTo(value, base).

DirectParamTriggered(invocation, actual) :-
  DirectRecursiveParameter(actual, type_parameter, method, invocation),
  Reachable(method),
  Composite(container),
  ValueType(value, container),
  VariablePointsTo(value, actual).
```

Listing 5.4: Rules for traversals and triggers

5.3.3 Selecting Libraries for Analysis

We evaluated the analysis on a set of 16 widely used Java parser libraries for different data formats (Table 5.1). These popular libraries are known to parse external data formats, and are therefore prone to the vulnerabilities we study. These libraries process data used in messaging, object serialisation and document representation, represented in various text and binary formats. We covered libraries for parsing or processing XML, JSON and YAML, PDF and external DSLs. These libraries are the standard and most widely used ones available for Java. Table 5.1 also contains usage data showing how many Maven artefacts depend on these libraries. This provides some indication of the impact vulnerabilities in these libraries have, based on usage statistics[6].

[6]Maven usage statistics (obtained on 12 Feb. 2020).

Table 5.1: Java libraries for analysis

Library	Input Format	Version	Usage
batik	SVG	1.1	115
gson	JSON	2.8.5	11,900
jackson	JSON	2.9.8	6,829
jettison	JSON	1.4.0	753
jfxrt	Java Image IO	1.8	N/A
mongo	BSON	3.9.1	1,048
mvel2	MVEL2	2.4.3	395
ognl	OGNL	3.2.10	339
pdfbox	PDF	2.0.12	403
pdfxstream	PDF	3.7.0	N/A
protobuf	Protocol	3.6.1	2,407
sanselan	Images	0.97	52
snakeyaml	YAML	1.23	1,962
stringtemplate	StringTemplate	3.2	270
xbean	SOAP/XMLBean	3.0.2	632
xstream	XStream	1.4.11.1	1,711

After analysing these 16 libraries, we proceeded to evaluate whether libraries for other languages, shown in Table 5.2, were vulnerable, using the payloads constructed. These libraries were selected based on availability and popularity. librsvg is used in the GNOME desktop[7], Ghostscript is widely deployed and used for processing PDFs, a popular vector-based illustration software - Inkscape uses librsvg and cairo[8]. Inkscape [9] is also used by ImageMagick for SVG processing. We looked at solutions for sanitising SVG files, and found that svg-sanitizer is the most widely used (e.g. WordPress, drupal).

5.3.4 Triggers or Entry Points

Identifying a trigger is a manual step that requires domain knowledge of the library under analysis. For example, a trigger for a data serialisation format could potentially be the deserialise method, which can be executed after the serialised format is read from disk. For image formats this could be the rasterisation or conversion process that would require traversals of the structure.

[7] `https://www.gnome.org`
[8] `https://cairosvg.org/`
[9] `https://inkscape.org/`

Table 5.2: Non-Java libraries investigated

Library	Language	Input Format	Version
Qt	C++	SVG	5.14.1
librsvg	Rust	SVG	2.46
PDFtk	GCJ	PDF	2.0.2
qpdf	C++	PDF	9.1.1
PDFium	C++	PDF	N/A
ghostscript	PostScript	PDF	9.25
svg-sanitizer	PHP	SVG	0.13.2
resvg	Rust	SVG	0.8.0
cairosvg	Python	SVG	2.4.2

Some libraries have command line interfaces which initiate calls to the trigger methods. For instance, Apache PDFBox. In the case where we analysed libraries without command line interfaces, a driver was required as an entry point for the input program for the pointer/call graph analysis. We have written custom drivers for libraries that are not bundled with a command line interface. The driver provides an entry point as well as a facility to interact with the library's API. In the case of SnakeYAML, the driver consists of statements to instantiate the parser and load a file as shown in Listing 5.5. Only MVEL2, PDFBox, Batik and PDFxStream come with built-in command line interfaces and did not require custom drivers.

```
package drivers.snakeyaml;

import org.yaml.snakeyaml.Yaml;
import org.yaml.snakeyaml.constructor.SafeConstructor;

public class Main {

    public static void main(String[] args) {
        Yaml yaml = new Yaml(new SafeConstructor());
        Object obj = new Object();
        String output = yaml.dump(obj);
        Object y = yaml.load(output);

    }
}
```

Listing 5.5: Driver for SnakeYAML

5.3.5 Evaluation

Static Analysis

The experiments were performed on an Intel(R) Core(TM) i7-8700 CPU @ 3.20GHz with 64GB of RAM on Linux Ubuntu 18.04.3. Doop[10] was run using the Java 8

[10]Doop version built from commit: https://bitbucket.org/yanniss/doop/commits/ba731a63c90e94f9e94afca39ff15c5082bf868d

platform as implemented in Oracle's version 8 of the JDK (build 1.8.0_221-b11). We used the following options for the analysis:

- analysis: `-a context-insensitive`
- main class: `-main` option was used with the driver class as an argument.

For each project, we extracted additional facts for type parameters for the fields in each class of the input library, and then executed DOOP with custom rules to compute:

- Composites (i.e., facts instantiating the Composite rule).
- Recursive methods (direct and indirect).
- Heap flows to refine the list of recursive methods, i.e. an object that is of composite type must flow to the parameter of a recursive method.
- Methods with heap flows from the previous step, and that are reachable via the entry point.

There are some limitations of the static analysis that degrade precision and recall. With regard to precision, we did not use a more precise context-sensitive pointer analysis on DOOP for performance reasons, as DOOP timed-out while analysing two libraries with the analysis options set to: `2-object-sensitive+heap` and the timeout at 90 minutes. The analysis could be boosted further by (1) adding more precision to the analysis, such as ruling out false positives by means of symbolic execution [8], or (2) by discovering more true positives automatically, e.g. by using fuzzing [75]. Both approaches are interesting topics for future research. Regarding recall, we expect that we miss potential vulnerabilties due to the presence of dynamic language features [106, 177]. We are aware of at least one related vulnerability that uses reflection (i.e., Pufferfish [58]). The obvious way to boost this is to either run a static analysis with full reflection support [168] (which we decided not to use due to the costly performance overhead that comes with it), or to run DOOP in hybrid mode with a dynamic pre-analysis [82] (or Soot with Tamiflex [27]), which however relies on the availability of a suitable driver.

Manual Evaluation of Analysis Results

At the end of the static analysis we find a set of candidate instances, i.e. bindings of the concepts used in TTT to concrete artefacts within the program under analysis. However, these may contain false positives as the malicious computation is effectively prevented by some program logic. While we cannot accurately eliminate false positives,

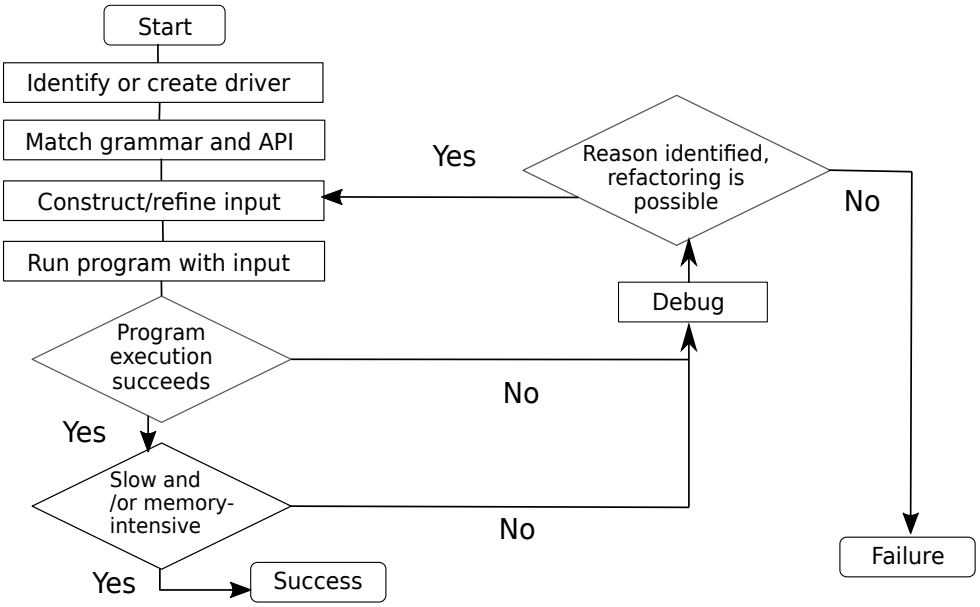


Figure 5.2: Workflow for manual evaluation of results

we conducted a manual step to identify true positives, by constructing payloads that expose the respective vulnerability.

We depict the systematic process that was followed in evaluating the results of the analysis in Figure 5.2. The first step was identifying a driver for the parser. In some cases we created the driver, as mentioned earlier. This involved identifying the trigger for initiating the recursive operation over the parsed structure. The next step was obtaining the results of the analysis, the list of recursive methods and composites and then matching the API of the composite and method to the grammar. There is work to automate this process in fuzzing grammars [90]. We then constructed an input to match with the portion of the grammar identified against the candidate API and executed the program with this input. If the execution succeeded we could determine whether or not the program's performance is sensitive to the structure of the input. If so, we had succeeded in constructing a potential payload. If this was not the case, we attempted to debug and identify a refactoring to the input that changes the behaviour. If there were no potential refactorings we ended the inspection.

In summary, the manual effort consisted of inspection of the program's source code, debugging and reviewing specifications against the implementation for the particular parser library.

The analysis found a list of methods with potential vulnerabilities. These may not always correspond to actual vulnerabilities if the program guards against invocations of the respective traversal methods and parameters triggering the traversal (for instance,

check if all elements only have a single parent.)

Table 5.3 shows a summary of the outcomes of the experiment, including the analysis run times for the 16 libraries. Methods and composites from the library and their dependencies are shown. The **Topology** column lists all composite types in the library. The **Methods (Recursive)** column lists all direct recursive methods. The **Methods (Composite)** column lists only those methods that have a composite type as a parameter. From these methods, the **Traversal** column lists methods that have a value flow within the composite field to the recursive callsite. The **Trigger** column lists reachable traversals for the identified topologies.

5.4 Results and Discussion

Table 5.3 lists the results of the analysis for direct recursion along with composites. The final column, which lists reachable traversals that are candidates for potential vulnerabilities. We used the inputs created from the investigation to detect vulnerabilities in non-Java libraries. In the following sections, we discuss some of the vulnerabilities detected in more detail.

Table 5.3: Overview of experiments (composites and recursion)

Library	Format	Time (sec)	Topology	Methods (Recursive)	Methods (Composite)	Traversal	Trigger
batik	SVG	505	430	971	595	34	34
gson	JSON	300	27	52	25	1	1
jackson	XML	404	126	635	167	10	10
jettison	JSON/XML	296	19	12	8	1	1
jfxrt	XML	784	0	2	0	0	0
mongo	BSON	433	275	507	224	0	0
mvel2	MVEL	173	93	320	135	4	2
ognl	OGNL	317	29	81	64	4	4
pdfbox	PDF	703	480	743	247	11	11
pdfxstream	PDF	334	115	209	118	3	3
protobuf	Protobuf	383	199	341	202	5	5
sanselan	Image	307	35	32	6	0	0
snakeyaml	YAML	307	30	35	13	3	3
stringtemplate	Template	306	32	31	18	0	0
xbean	XML	492	363	513	230	0	0
xstream	XML	331	120	218	100	1	1

5.4.1 PDF Vulnerabilities

The analysis detected 11 reachable traversals that recurse on a parameter in the PDFBox library. From these results, a vulnerability was confirmed in Apache PDFBox, the most used Java PDF library in the Maven repository[11].

```
public class COSDictionary extends COSBase
{
     // The name-value pairs of this dictionary.
    protected Map<COSName, COSBase> items =
    new LinkedHashMap<COSName, COSBase>();
...

}
...
private int checkPagesDictionary(
    COSDictionary pagesDict,
    Set<COSObject> set)
{
  // check for kids
  COSBase kids =
    pagesDict.getDictionaryObject(COSName.KIDS);
  int numberOfPages = 0;
  COSArray kidsArray = (COSArray) kids;
  List<? extends COSBase> kidsList =
    kidsArray.toList();
  for (COSBase kid : kidsList)
  {

  COSBase kidBaseobject = kidObject.getObject();
  COSDictionary kidDictionary =
    (COSDictionary) kidBaseobject;
  COSName type =
    kidDictionary.getCOSName(COSName.TYPE);
  if (COSName.PAGES.equals(type))
  {
  // process nested pages dictionaries
  set.add(kidObject);
 numberOfPages +=
    checkPagesDictionary(kidDictionary, set);
  }

   }
...
 }
```

Listing 5.6: Composite and recursion for PDFBox

The PDF document format, Carousel Object Structure (COS), is described in the PDF Reference [138]. It supports basic types such as booleans, integers, real numbers, strings, names and more crucially, arrays and dictionaries. The particular composite topology in the library consists of `COSDictionary` as the container and `COSBase` (shown in Listing 5.6) as the component where the children are stored in an object that implements the `Map` interface.

The recursive method that traverses this structure is `checkPagesDictionary(COSDictionary pagesDict)` defined in `org.apache.pdfbox.pdfparser.COSParser`, which

[11] `https://mvnrepository.com/`

is invoked when the PDF file is parsed. The only constraint in the path condition from the entry into the method to the recursive call is the presence of child objects that are of the type `COSName.PAGES`.

Manual inspection of the source code and the PDF specification [138] revealed that the root of a PDF document, the catalog, points to a dictionary referred to as a *Page Tree*, which can in turn refer to another *Page Tree*. This structure parses to a `COSDictionary` composite and we can craft a PDF document that parses into an object graph with the many-to-many pattern.

Passing the crafted PDF to the application revealed that it can result in attacks on responsiveness and disk space as the application can also be used to convert the pages in a PDF to disk as images. The issue was reported and it has been accepted with the identifier CVE-2018-11797.

The same PDF document was used to confirm the vulnerability in PDFxStream (CVE-2019-17063), and it also revealed the same vulnerability in PDFtk[12], the PDF toolkit. They both use a `HashMap` to store the COS structure, which in principle makes a DoS attack possible as in SerialDoS.

We also tested the PDF document on Ghostscript [73], a PostScript and PDF interpreter. Using the crafted PDF as an input to Ghostscript resulted in DoS. This bug was accepted as a security vulnerability (CVE-2018-19478). The PDF parser in Ghostscript is implemented in PostScript and the traversal of the COS was for an entirely different purpose when compared to the previous cases, which was to detect cycles in the *Page Tree* that had caused a security vulnerability in Ghostscript. This suggests that traversals of the form that we have studied occur across multiple languages.

5.4.2 Scalable Vector Graphics (SVG) Vulnerability

SVG [181] is an XML-based vector format for two-dimensional graphics with support for interactivity and animation. It is supported by all browsers and is used in illustration programs such as Adobe Illustrator and Inkscape. SVG is processed by the Batik library (the de facto standard for SVG processing in Java). The analysis reported 34 reachable traversals for the library. One particular composite topology in the library consists of `Node` (shown in Listing 5.7) as the container with children or parents of the same type.

The recursive method that traverses this structure is `String getCascadedXMLBase (Node node)` defined in `org.apache.batik.anim.dom.SVGOMElement`, which is invoked when the SVG file is parsed. The only constraint in the path condition from the entry into the method to the recursive call is the presence of child objects that are of the same node type as the passed parameter (i.e. XML element nodes).

[12] `https://www.pdflabs.com/tools/pdftk-the-pdf-toolkit/`

```
public interface Node {
...
    public Node getParentNode();
...
}

protected String getCascadedXMLBase(Node node) {
  String base = null;
  Node n = node.getParentNode();
  while (n != null) {
  if (n.getNodeType() == Node.ELEMENT_NODE) {
  base = getCascadedXMLBase(n);

    break;
    ...
    }
    ...
}
```

Listing 5.7: Composite and recursion for batik

Any SVG graphics element is potentially a template object that can be re-used (i.e., instanced) in the SVG document via a `<use>` element. The `<use>` element references another element and indicates that the graphical contents of that element is included/-drawn at that given point in the document. The `<g>` element can be used to specify a grouped container of elements. The `<g>` element in conjunction with the `<use>` element can be used to construct a nested structure to trigger the detected vulnerability. The `<use>` element can also be used to construct the SVG version for SerialDoS as shown in Listing 5.8. We based the SVG file on this ability to nest references using `<g>` and `<use>` elements. There is an additional way to reference elements, as shown in Listing 5.9, which uses the pattern tag and its fill attribute set to a `url` function containing the reference id for an element in the document.

```
<g id="t0a">
<use xlink:href="#t1a"/>
<use xlink:href="#t1b"/>
</g>

<g id="t0b">
<use xlink:href="#t1a"/>
<use xlink:href="#t1b"/>
</g>
```

Listing 5.8: Nested references in SVG

```
<pattern id="h" ... >
      <rect  fill="url(#g)" stroke="green" />
```

Listing 5.9: References with use() function in SVG

The same SVG document was used to verify vulnerabilities in web browsers, and a core Linux SVG rendering library (librsvg[13]). We reported the issue for librsvg, which was fixed by the vendor and a CVE was obtained (CVE-2019-20446). We found

[13] `https://wiki.gnome.org/action/show/Projects/LibRsvg`

the crafted file to impact all tested browsers (e.g. Mozilla Firefox (version 73.0), Google Chrome (Version 77.0.3865.120, Official Build) by excessively consuming resources (memory and CPU) for Firefox and crashing the active browser tab in Chrome. This can be used by malicious parties to craft client-DoS for websites that allow links to SVG code in user input (e.g. Markdown with links to external SVG files in user comments). We confirmed this observation for the StackOverflow[14] Q&A platform, GitLab[15] and GitHub[16] issue trackers. The impact on these services is that they can render the page inaccessible to users if it has malicious SVG content. We also considered svg-sanitizer[17], which performs server-side sanitization of SVG content. On passing the crafted SVG file as input, svg-sanitizer, entered a non-terminating computation. This is an example of a mitigation measure becoming an attack surface. It is available as a plugin for WordPress and Drupal, and using vulnerable versions of the plugin can be make the services susceptible to DoS attacks.

5.4.3 YAML Vulnerability

YAML is a popular and widely used (human readable) serialisation language for data interchange and application configuration. It supports primitives and common data structures such as maps and lists [195]. We looked at the SnakeYAML library in Java and the analysis reported three reachable traversals, one of which is shown in Listing 5.10. The composite is `MappingNode` with a list of `NodeTuple` as children that can potentially have the same type as the parent. The analysis shows a flow from the list to the first parameter at the highlighted recursive call site.

```
public class MappingNode extends CollectionNode<NodeTuple> {
  private List<NodeTuple> value;
    ...
}

private List<NodeTuple> mergeNode(MappingNode node, boolean isPreffered,
  Map<Object, Integer> key2index, List<NodeTuple> values) {
    Iterator<NodeTuple> iter = node.getValue().iterator();
    while (iter.hasNext()) {
      final NodeTuple nodeTuple = iter.next();
      final Node keyNode = nodeTuple.getKeyNode();
      final Node valueNode = nodeTuple.getValueNode();
      if (keyNode.getTag().equals(Tag.MERGE)) {
        iter.remove();
        switch (valueNode.getNodeId()) {
          case mapping:
            MappingNode mn = (MappingNode) valueNode;
            mergeNode(mn, false, key2index, values);

  ...
```

[14] https://stackoverflow.com
[15] https://gitlab.com
[16] https://github.com
[17] https://github.com/darylldoyle/svg-sanitizer

```
}
```

Listing 5.10: Composite and recursion for SnakeYAML

The code in Listing 5.10 is the implementation of the << merge key feature in YAML, which is used to indicate that all the keys of one or more specified maps should be inserted into the current map. If the value associated with the key is a single map, each of its key/value pairs is inserted into the current map, unless the key already exists in it. If the value associated with the merge key is a sequence, then this sequence is expected to contain multiple maps and each of these are merged in order. Listing 5.11 shows the use of of YAML merge as well as the use of YAML aliases in constructing SerialDoS type inputs, which were detected as vulnerabilities by the analysis.

```
? - &t2a
    - &t3a [lol]
    - &t3b [lol]
  - &t2b
    - *t3a
    - *t3b
: value
--
{ << { << { key: value} } }
```

Listing 5.11: Merging map keys in YAML

We created a YAML file with nested merges and nested lists with aliases forming the topology, and passed the file as input to the SnakeYAML driver to confirm that it crashed from stack exhaustion for the nested merges case, and entered a long-running computation for nested lists. Consequently, this issue has been reported to the vendor. In a recent release a feature has been added to the library to optionally disable alias processing.

5.5 Performance Measurements

In order to study the performance characteristics of these vulnerabilities, we created a benchmark based setup to create and parse SVG, PDF and YAML documents with different nesting depths in the document structure. We wrote a driver to parse these documents with the Apache Batik, PDFBox and SnakeYaml libraries, and executed them to measure the runtime and memory allocation of the parser using JMH [93], with five warm up iterations and five measurement iterations. We measured average running time, and used a memory profiler to show memory allocation rates.

The size of the inputs, in bytes, against the depth of the structure is shown in Fig. 5.3. The results for resource usage are shown in Fig. 5.4 and 5.5. The measurements depict exponential increases over resource consumption, both time and memory, as the depth increases linearly for the SVG and PDF inputs. For YAML, only time consumed grows exponentially as the recursive method is not resource-monotonic.

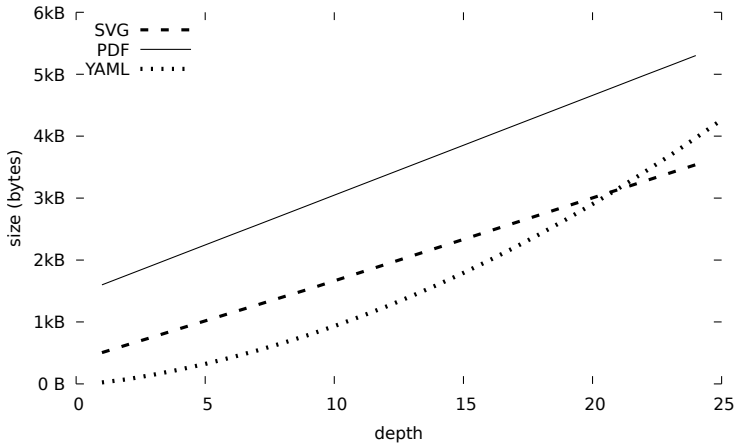


Figure 5.3: Input sizes for different depths

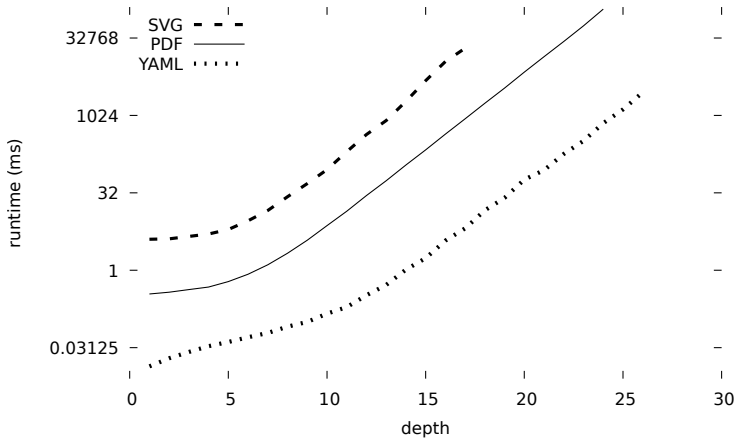


Figure 5.4: Runtime performance for documents with different depths

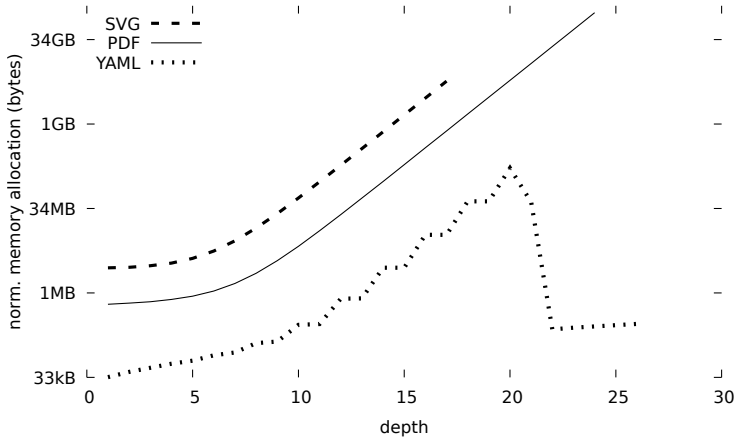


Figure 5.5: Memory performance for documents with different depths

5.6 Limitations

Our implementation of the topology (composite) pattern is based on standard container types available in the standard library of the language. Hence, we may miss custom data structures or non-standard containers with many-to-many relationships (e.g., pairs, data structures for binary trees). But the usage of such custom containers is uncommon in Java programs due to a focus on reuse, the availability of a mature standard library and the fact that third-party libraries that provide extra container data structures (e.g., google guava, apache-commons) use the same standard library interfaces, and are therefore supported by the analysis. The analysis is applicable to any external data format that is parsed to composite structures and uses recursive functions for traversals, however the manual construction of exploits is simpler for text-based formats and formats where the task of understanding of an implementation is well-informed by a specification.

5.7 Conclusion

We presented an approach to classify and detect a class of DoS vulnerabilities in parsing data structures. We evaluated this approach on a set of 16 Java parser libraries, with a Datalog-based formulation of a static analysis, using the Doop analysis framework for Java. The results revealed four new vulnerabilities in widely used Java PDF, SVG and YAML libraries. A further evaluation also revealed seven more vulnerabilities in parser libraries for Rust, PHP, C++ and PostScript. From these reports, we have obtained four CVEs and reported a total of 11 security issues to vendors (7 of which have been accepted). The results confirm that a lightweight static analysis can be useful in uncovering vulnerabilities that belong to this class.

Chapter 6

Non-Java Languages

We have seen DoS vulnerabilities based on serialisation in the Java programming language, and we now seek to investigate the case for other programming languages. The focal point of the investigation is YAML, a widely used data serialisation language, which enables us to study parsers in multiple languages using a common format. From among the other options, i.e., JSON and XML, JSON is not as expressive, and XML parser vulnerabilities have already been studied. Further, YAML injection vulnerabilities have been reported but DoS vulnerabilities have not received as much attention. Another reason for choosing YAML is that it is a common format that is popular for application configuration, data exchange and it is supported in most widely-used programming languages.

Language-agnostic deserialisation formats such as YAML, JSON and XML are generally seen as safe choices. Among these formats, YAML (YAML Ain't Markup Language) [195] is widely used in data interchange, serialisation, and application/environment configuration. YAML is essentially a superset of JSON [94], with a lightweight syntax that is optimised for human readability and hand-editing. It has a type system comprised of scalars (numbers, strings and booleans) and collections (lists and maps.) Furthermore, it can support arbitrary types from the parser library's environment through "type tags". YAML is a cornerstone in IT environments, where automation in development and deployment is largely driven by configuration files[1] that are either in YAML or JSON format. Examples of its use in this setting are for specifying APIs according to the OpenAPI specification [127], Continuous Integration/Delivery (CI/CD) scripts, and describing cloud infrastructure as in Amazon's AWS CloudFormation templates [18].

Given its ubiquity, it is desirable that parsing YAML is safe and secure. However, this is not the case. Several remote code execution vulnerabilities in deserialising YAML data have been reported since 2013 for multiple languages, e.g. CVE-2018-13043 [146]

[1] `https://www.theregister.co.uk/2018/11/19/popular_programming_language_yaml/`

and CVE-2017-18342 [147]. Remote code execution vulnerabilities in YAML parsing libraries have affected several applications, examples being Swagger, Apache Brooklyn and Apache Camel [182] that use the SnakeYAML library [170]. The prevailing advice[2] is to use the safe load feature in these libraries, as unsafe loading can be used to instantiate types with data from YAML input using its tag feature. Safe load limits the classes that are deserialised to the basic types available in YAML and rejects user-defined types from being deserialised.

DoS attacks have been extensively studied for XML parsers [171], in particular, vulnerabilities that result in exponential or quadratic blowups in resource usage while parsing input. There has been less attention on DoS attacks on YAML parsers.

We study DoS vulnerabilities for 14 YAML processing libraries (listed in Table 6.2) for ten of the most widely-used languages, including PHP, Java and JavaScript. As a result of this study we have discovered seven previously unknown vulnerabilities that can be used to launch DoS attacks on applications that use these YAML libraries. We have also made the results of this work reproducible in a public repository[3].

We begin with a basic introduction to YAML. We examine the characteristics of the DoS vulnerabilities in this study and then evaluate YAML libraries and discuss the results of the evaluation.

6.1 YAML Basics

In this section we will cover YAML features, as documented in the YAML specification [195], that are relevant to the vulnerabilities in this study. YAML is a superset of JSON, both of which are human readable data interchange formats. While JSON's foremost priority is simplicity, YAML attempts to be more human readable by relying on white space for structure, and it supports a richer information model by allowing application defined types and object references.

6.1.1 Types

YAML data structures can be built with three basic primitives: a mapping - hashes/dictionaries that are unordered association of unique key to value mappings; a sequence - arrays/lists that are ordered; and scalars - numbers, strings, date/timestamps, booleans and null. These primitive types map to similar structures common in scripting languages such as Perl, Python, PHP, Ruby, and JavaScript. Repeated objects can be identified at the first occurrence by a named anchor and subsequently referenced one or more times in the document. In the second version of the specification, types

[2] `https://security.openstack.org/guidelines/dg_avoid-dangerous-input-parsing-libraries.html`

[3] `https://bitbucket.org/unshorn/dosyamlpub`

are referred to as schemas. The "Failsafe" schema, as described in the specification [195], supports only sequences, maps and strings. The JSON schema offers maximum support for JSON. A YAML library can also implement a language-specific schema, with tags for the language's standard or user-defined types.

6.1.2 Syntax

YAML uses indentation (block style) or explicit JSON-like syntax (flow style) to denote structure. In block style, hierarchy in the document structure is determined using indentation (double spaces). In block style, sequences are separated by lines and each item is prefixed by a dash. Mappings are represented similarly, with keys and values separated by a colon. Mappings and sequences can be arbitrarily nested. The language feature to reuse constructed object instances are anchors and aliases. Anchors which name an element (a scalar, a mapping or a sequence) are written as `&NAME` where the item identified by the string, `NAME`, can be referred to with the alias `*NAME`.

Another distinguishing feature of YAML, unlike JSON, is that it supports complex mappings. Whereas JSON maps can only have numbers or strings as keys, YAML maps can have keys of an arbitrary type such as a sequence, a mapping, or an alias. A single YAML file can also contain multiple documents seperated by the marker `---`.

Listing 6.1 shows an example YAML file, consisting of three documents separated by `---`. The first document is a single map with a complex key, which begins with the syntactic marker "?". The key `[item1, item2]` is a list that maps to the string "value". In the second document, `var` is an anchor for the string value "foo", and the second item is a nested list that refers to the same value with the alias `*var`. The entire document in string form is `[foo, [foo]]`. While the previous examples use the Failsafe schema, the third document is an example of a language specific schema with a type tag for a Java class, `com.example.Customer` and parameters for its constructor. The YAML processor will instantiate this object when the file is loaded.

```
1 ?
2 - item1
3 - item2
4 : value
5 ---
6 - &var foo
7 - - *var
8 ---
9 !!com.example.Customer
10 firstName: "Rick"
11 lastName: "Astley"
```

Listing 6.1: Example YAML documents

Processing YAML involves three stages. In the case of loading a YAML file, these are parsing the presentation stream, composing a representation graph and, finally, constructing native data structures from the YAML representation.

6.2 DoS Vulnerabilities

In the scope of libraries that process YAML, we look for methods that can be triggered while the parsed YAML structure is being converted to native data types and that can then exhaust resources during this conversion. The vulnerabilities have the characteristics discussed earlier in Chapter 3.

Maps are often implemented as hash tables. Since YAML supports complex keys, i.e., keys that are types other than plain numbers and strings, the conversion from YAML to the maps must trigger the computation of hash values for lists or maps that are used as keys. These computations can involve the recursive traversal of elements within them. For example, the hash value of a list may be computed based on the hash values of its elements.

In the example in Listing 6.2, the key is a list with a sole element that is a reference to itself. As the hash computation can be recursive on the elements in the list, the call tree for this computation is an infinite chain of method invocations that abnormally terminates from call stack exhaustion.

```
1 ? &id001
2 - *id001
3 : value
4 ---
5 ? &id001
6 - - *id001
7 : value
```

Listing 6.2: Loops and cycles in list as key

Listing 6.3 is the YAML equivalent of XML "billion laughs" implemented as nested lists. A similar expansion of the structure to a string will occur when it is stringified (e.g. using the `toString()` method in Java.)

For clarity, we ignore internal structure within the objects in their language-specific implementations. In this case, the depth of this structure is two. Note the number of objects created is seven, which is the two strings, the two lists enclosing them, the two inner lists and the outer list.

When the list is converted to a string by triggering the outer list's `toString()` method, it sets off a recursive call to `toString()` on each of its children in the object graph. As discussed in Chapter 3, DoS vulnerabilities, the number of invocations is exponential in depth of the object graph. The first snippet, nested to a depth of two, expands to the list `[[[lol], [lol]], [[lol], [lol]]]` if the list is converted to a string. At depth three it doubles in size to `[[[[lol], [lol]], [[lol], [lol]]], [[[lol], [lol]], [[lol], [lol]]]]`.

In the second document in the listing, the nested list is used as a complex key with the value "value". As the insertion of the value into the map by the parser will require computing the hash value for the nested list, it will trigger the hash function on the

outer list. The rest of the invocations are similar to the analysis for stringification. The notable difference is that the computation does not result in memory exhaustion because, unlike the string computation which concatenates and expands the string with each invocation, the accumulated hash value is a fixed size integer.

A additional but key requirement is that the depth of the object graph must be limited to avoid a stack overflow, yet deep enough to cause memory or CPU resource exhaustion.

```
1
2 - &id2A
3   - &id3A [lol]
4   - &id3B [lol]
5 - &id2B
6   - *id3A
7   - *id3B
8
9 ---
10 ? - &id2A
11     - &id3A [lol]
12     - &id3B [lol]
13   - &id2B
14     - *id3A
15     - *id3B
16 : value
```

Listing 6.3: Nested lists

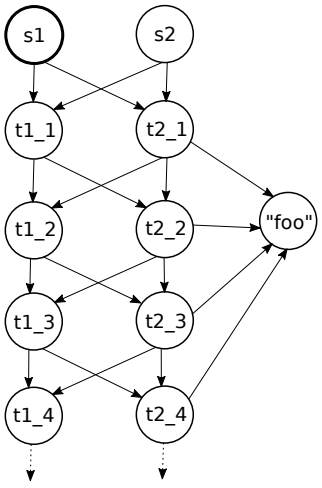


Figure 6.1: Graph representation of list

.

Table 6.1: Character count for test vectors

	Loop	**Cycle**	**Nested lists**
Test vectors 1 (as key in map)	24	26	12,182
Test vectors 2 (as value in map)	21	23	11,783
Test vectors 3 (as element in list)	20	22	12,174

6.3 Experimental Setup

The experiment was conducted on an Intel i5 2.40 GHz machine with 16GB memory and Linux Ubuntu 18.04.1, within a Docker container to improve reproducibility. We generated test vectors and wrote drivers programs for each library, then the drivers were executed against each of the test vectors. Each execution was set to timeout after two minutes, and the exit code was recorded. Cases where the exit code was non-zero, i.e., the execution either timed-out or failed, were examined in more detail to determine the behaviour of the library.

6.3.1 Test Vectors

We created nine test vectors designed to trigger DoS attacks. The nine test vectors, as listed in Table 6.1, are based on three structures where each structure is instantiated with three patterns (pattern in key of a map, value of a map and as an element of a list). The patterns are based on recursive data. The two recursive data patterns are a list with a reference to itself constituting a loop; a list nested within another with a reference to its parent forming a cycle. They are both implemented as in Listing 6.2. The third is a deeply nested list as described in Section 6.2. The character counts listed in the table show that each of the vectors are small at under ten kilobytes in file size at most and hence, YAML libraries should be expected to parse them reasonably fast (within seconds.) In the case of the deeply nested list, there are 98 references/aliases in the test vectors. The nesting depth was set at 50, exercising care that the depth would not exhaust the stack before triggering a potential long-running computation. Exhausting the stack would result in an immediate stack overflow error whereas the longer computation would have a bigger impact by targetting either memory or CPU resources.

All nine data files for these test vectors were generated using the Java SnakeYAML [170] library and the files were verified for correctness by parsing them with the LibYAML parser for C [104], which is the YAML reference parser implementation that forms the basis for many YAML libraries (e.g. Python YAML [152], Rust Yaml [157].)

Table 6.2: Analysed libraries

Language	Library	Version	Lang. version
JavaScript	js-yaml	3.12.1	NodeJS 8.10.0
PHP	php-yaml	2.0.4	PHP 7.2.15
	Symfony Yaml	4.2.0	
Python	PyYAML	3.13	Python 3.6.7
	ruamel.yaml	0.15.89	
Dart	yaml	2.1.15	Dart VM 2.1.1
C#	YamlDotNet	5.2.1	Mono 4.6.2
Perl	YAML:Syck	1.31	Perl 5, version 26
	YAML::XS	0.76	
	YAML	1.27	
Ruby	psych	3.1.0	2.5.1
Java	SnakeYAML	1.23	OpenJDK 1.8.0_191
Rust	yaml-rust	0.4	Rust 1.32
Swift	YamlSwift	3.4.3	4.2.3

6.3.2 Library Selection

We selected the YAML processing libraries from the list available at the official YAML website[4]. We confirmed the popularity of the libraries on package management repositories (e.g. NuGet[5] for C#, npm[6] for JavaScript and CPAN[7] for Perl) to ensure that these libraries were the most widely used in their respective language communities. If multiple libraries exhibited similar popularity, such as in the case of Perl, we included all of them. In some cases where data format parsers (e.g. Jackson for Java which depends on SnakeYAML for parsing YAML) depended on a library that is already in the list, we did not include it. We also excluded low level parser libraries such as LibYAML [104] from the study, as our focus is only on libraries that parse and deserialise YAML to their respective language's native data types. We have also excluded libraries that only perform type-specific deserialisation (e.g. *serde-yaml* [165] and *go-yaml* [80], for Rust and the Go language.)

6.4 Results

Tables 6.3, 6.4 and 6.5 lists a summary of the results from the experiment for the test vectors, *Loop*, *Cycle* and *Nested aliases*. *Success* indicates the driver terminated without errors, *Timeout* indicates that the driver timed-out after the threshold of two minutes, *Error* indicates that the driver exited with an error, which is parsing failure,

[4] `http://yaml.org`
[5] `https://www.nuget.org/`
[6] `http://www.npmjs.com`
[7] `https://www.cpan.org/`

Table 6.3: Library exit codes for test vectors 1: structure as key in map

Library	**Loop**	**Cycle**	**Nested lists**
JavaScript (Node)	Success	Success	Error
PHP PECL Yaml	Success	Success	Success
PHP Symfony	Error	Error	Error
Python YAML	Error	Error	Error
Python ruamel.yaml	Success	Error	Error
Dart	Stack overflow	Stack overflow	Timeout
C#	Stack overflow	Stack overflow	Timeout
Perl Syck	Error	Error	Success
Perl XS	Success	Success	Success
Perl YAML	Error	Error	Error
Ruby	Success	Success	Timeout
Java	Stack overflow	Stack overflow	Timeout
Rust	Success	Success	Out of memory
Swift	Error	Error	Timeout

and *Out of memory* indicates that the driver abnormally terminated from a failure to allocate memory.

JavaScript

For the tests, the *js-yaml* [96] YAML library was executed in the NodeJS environment. It failed on the nested structure as a key test with the error: `RangeError: Invalid string length` in the *Array.toString* method. Further investigation revealed that *js-yaml* converts keys (keys in the parsed YAML could be objects other than strings or numbers) to strings, as JavaScript objects support only string and numeric values in maps.

We investigated with nesting levels lower than 50, and with a structure of depth 30, we were able to crash NodeJS with an out of memory error confirming that DoS via memory exhaustion is possible for the library. The vulnerability has been reported to the npm registry for JavaScript packages and it has been fixed[8]. We also reproduced the same results on a browser (Google Chrome) with *js-yaml*.

PHP

The PHP Yaml Extension [144] succeeded in all the tests. Since PHP does not support non-string/numeric keys in its primary data structure, associative arrays, we found that it stringifies complex keys. In contrast to *js-yaml*'s approach, instead of the full string representation of the structure it uses PHP's default serialisation format for the key. As PHP's serialisation supports internal references [143] [55] this results in a compact

[8] https://www.npmjs.com/advisories/788

Table 6.4: Library exit codes for test vectors 2: structure as value in map

Library	**Loop**	**Cycle**	**Nested lists**
JavaScript (Node)	Success	Success	Success
PHP PECL Yaml	Success	Success	Success
PHP Symfony	Error	Success	Success
Python YAML	Success	Success	Success
Python ruamel.yaml	Success	Success	Success
Dart	Success	Success	Success
C#	Success	Success	Success
Perl Syck	Success	Success	Success
Perl XS	Success	Success	Success
Perl YAML	Success	Success	Error
Ruby	Success	Success	Success
Java	Success	Success	Success
Rust	Success	Success	Out of memory
Swift	Error	Error	Error

Table 6.5: Library exit codes for test vectors 3: structure as item in list

Library	**Loop**	**Cycle**	**Nested lists**
JavaScript (Node)	Success	Success	Success
PHP PECL Yaml	Success	Success	Success
PHP Symfony	Error	Success	Success
Python YAML	Success	Success	Success
Python ruamel.yaml	Success	Success	Success
Dart	Success	Success	Success
C#	Success	Success	Success
Perl Syck	Success	Success	Success
Perl XS	Success	Success	Success
Perl YAML	Success	Success	Error
Ruby	Success	Success	Success
Java	Success	Success	Success
Rust	Success	Success	Out of memory
Swift	Error	Error	Success

representation if the key is a data structure that's compatible with PHP. If so, the compact key representation does not cause a time-out or memory exhaustion.

We experimented with variants of the YAML structure to create a new test vector that results in exponential growth of the serialised representation and we were able to cause memory exhaustion for the PHP YAML Extension. The key idea is to introduce a list with multiple complex keys at each nesting level so that a composite key cannot be built by one-off serialisation but instead serialisation must be invoked at inner levels. E.g. `{[{[a]: a}]: b}` where the key is not a valid PHP structure and the inner key must be serialised before the outer key. The second library, PHP Symfony [148], that we studied does not support complex keys.

We have reported the issue for the PHP PECL Yaml and it has been accepted as a security bug[9], pending a CVE after a fix is available.

Python

Python does not support complex keys, as the language does not allow mutable structures such as lists and maps as keys in dictionaries. With this limitation, the Python YAML libraries are not compliant with the YAML specification.

Dart

Dart successfully processes test vectors 2 and 3. On test vectors 1, structures as keys, Dart throws stack overflow errors for keys that have cyclic references and it timed out on the nested lists case.

C#

The results were similar to Dart. In the case of test vectors 1, the C# runtime exited with a stack overflow for the recursive structures case, and timed out for nested lists as a key.

Perl

The Perl XS [141] and Sycke [140] libraries use a fixed length string representation for complex keys and do not cause DoS. Otherwise the errors were due to parsing issues.

Java

For test vectors 1, the Java driver reported a stack overflow for the recursive structures, and timed out for the nested lists case on hash computation. We have reported the

[9] `https://bugs.php.net/bug.php?id=77720`

bug and it has been accepted as an issue[10] pending a fix.

Ruby

Ruby times out on test vectors 1/Nested lists while computing hashes. It does not cause a stack overflow as hash computation returns a special constant if recursion is detected.

Rust

The operating system kills Rust due to memory exhaustion for all three test vectors for the nested lists case.

Swift

The Swift YAML library causes errors in all but one test. The errors are due to parsing and for test vectors 1/Nested lists, it times out.

6.5 Discussion

Type-specific parsers and restrictions on YAML features offer a level of protection in the case of some libraries, although type-specific deserialisation might also not be completely impervious to attacks[11]. It can be difficult to strike a correct balance between usability and security. Earlier DoS vulnerabilities, e.g. CVE-2012-1152 [145], have been due to invalid input. Standard libraries could implement hash functions for containers that use dynamic programming to avoid redundant computation of elements within them.

In string serialisation (`toString` in Java and JavaScript) there is no check on the unbounded accumulation of the string representation of nested data structures like Arrays. In the .NET framework, collection classes do not override the `ToString` method[12] from `Object`, making string serialisation impossible by default, and leaving it to users to implement based on their specific use cases.

In the case of XML parsers, disabling inline document type definitions (DTD's) [178], which makes it impossible to define entities, is a protection against memory exhaustion while parsing user-supplied XML. Another measure for XML is to limit the expansion of recursively defined entities [178].

[10] `https://bitbucket.org/asomov/snakeyaml/issues/432/aggressive-yaml-anchors-causing`
[11] https://github.com/dtolnay/serde-yaml/issues/53
[12] `https://docs.microsoft.com/en-us/dotnet/api/system.collections.arraylist?view=netframework-4.7.2`

6.6 Native Serialisation in Other languages

In this section we investigate whether the vulnerabilities discussed above can be ported to other languages. We included C# as a language that is conceptually close to Java. It uses a similar type system and deployment model based on bytecode. We also looked into the portability of the identified vulnerabilities to a popular dynamic language, Ruby and a scripting language, JavaScript.

6.6.1 C#

We conducted experiments on both .NET 4.5 and Mono 4.6.1. The results were consistent for both implementations.

.NET offers several serialisation mechanisms, including `XML` and binary serialisation. .NET has separate generic and non-generic collections, the non-generic collection types in the namespace `System.Collections` include `Hashtable` and `ArrayList`, while the generic types in the namespace `System.Collections.Generic` include `HashSet<T>` and `LinkedList<T>`. The methods to establish equality and compute the hash code of collections are delegated to special *comparer* objects defined by the interface `System.Collections.IEqualityComparer` and its generic counterpart. This facilitates the implementation of collections with alternative comparison semantics, such as identity maps. Comparers are serialisable.

The deserialisation of `Hashtable` objects triggers the execution of `HashCode()` defined in the comparer being used, and nested containers are supported by all collection types and arrays. The behaviour of the hash calculation depends on the comparer being used. From the comparers available in the standard library, `HashSetEqualityComparator` used with nested (generic) hash sets did not exhibit the behaviour necessary to construct a `HashCode` call chain down the nested containers. We believe that this is actually due to a bug in .NET, due to a broken contract between `Equals` and `GetHashCode` in this class. This bug was reported and accepted[13]. However, constructing a non-generic `Hashtable` with a `StructuralEqualityComparer` results in recursive calls to `HashCode()` as expected, and can therefore be used to port the vulnerabilities. The code is shown in Listings 6.4 and 6.5, respectively.

Unlike the Java implementation of collection types, `ToString` for containers is not overridden. Therefore, we did not succeed in porting the vulnerability for an attack to exhaust heap memory .

```
1 using System;
2 using System.Collections;
3 using System.Runtime.Serialization;
4 using System.IO;
5 using System.Runtime.Serialization.Formatters.Binary;
```

[13] `https://github.com/dotnet/corefx/issues/12560`

```
6
7 public class SerialDOS {
8
9   public static void Main(){
10    //serialize
11    var outStream = new MemoryStream();
12    var bf = new BinaryFormatter();
13    bf.Serialize(outStream, payload());
14    //deserialize
15    var inStream = new MemoryStream(outStream.ToArray());
16    var deserializedObject = bf.Deserialize(inStream);
17  }
18
19  public static Object payload() {
20    var top = new object[2];
21    var comp = StructuralComparisons.StructuralEqualityComparer;
22    var root = new Hashtable(comp);
23    root.Add(top, "foo");
24    var s1 = top;
25    var s2 = new object[2];
26    for (int i = 0; i < 50; i++) {
27      var t1 = new object[2]; var t2 = new object[2];
28      s1[0] = t1; s1[1] = t2;
29      s2[0] = t1; s2[1] = t2;
30      s1 = t1; s2 = t2;
31    }
32    return root;
33  }
34 }
```

Listing 6.4: .NET/C# SerialDOS

```
1 public static Object payload() {
2 var top = new object[1];
3 var comp = StructuralComparisons.StructuralEqualityComparer;
4 var root = new Hashtable(comp);
5 root.Add(top,"");
6 top[0]=top;
7 return root;
8 }
```

Listing 6.5: .NET/C# Turtles all the way down (payload construction only)

6.6.2 Ruby and JavaScript

In collaboration with colleagues, we have also studied this class of DoS attacks with recursive or cross-referencing data structures for Ruby and JavaScript. This is reported in [58].

Ruby

In the case of Ruby, we evaluated MRI Ruby 2.0.0p648 and JRuby 9.1.6.0. Ruby has several serialisation mechanisms in addition to `YAML`, such as `Marshal` and `JSON`. Deserialisation of hash maps also triggers the execution of `hash`, and nested containers are supported. In contrast to Java, `hash` is executed in a controlled environment that

prevents recursion[14]. If recursion is detected, a special constant value is returned.

Another difference is that when containers are converted to strings using the Ruby stringify method (`to_s`) it does not concatenate elements to a larger string.

A similar, serialisation-related vulnerability was discovered and reported in 2013 [43]. Using this vulnerability it was possible to initiate a DoS attack by using a crafted JSON document to create a large number of symbols which were never garbage collected. In response to this, the garbage collector in newer versions of Ruby also collects symbols[15].

JavaScript

We evaluated JavaScript using the node.js v0.12.7 implementation. The version of JavaScript that was widely supported at the time of the study, standardised as ECMA-262, was 5 [59]. JavaScript provides a serialisation mechanism with the built-in JSON object [59, sect.15.12]. JavaScript 5 has no explicit support for maps or similar data structures in its type system [59, sect.8], and the `Object` type [59, sect.8.6] is used to represent map-like structures. The consequence of this is that only strings are allowed as keys in maps. JavaScript 6 adds support for proper maps that allow arbitrary ECMAScript language values (including objects) as both keys and values [60, sect.23.1]. However, the JSON serialiser does not serialise maps. For instance, evaluating the script in Listing 6.6 produces an empty string.

```
1 var map = new Map();
2 map.set('foo', 42);
3 var serMap = JSON.stringify(map);
4 // will output "{}"
5 console.log(serMap);
```

Listing 6.6: JavaScript 6 maps are not serialised

The semantics of JavaScript 6 maps is similar to identity maps in Java in the sense that it is not based on user-defined equality [60, sect.7.2.10]. While the standard stipulates that the "Map object must be implemented using either hash tables or other mechanisms that, on average, provide access times that are sublinear on the number of elements in the collection." [60, sect.23.1]. Such a hash function would be an implementation-specific system hash consistent with the built-in equality of objects. Therefore, JavaScript 6 does not provide recursive hash functions that can be exploited.

The JSON serialisation mechanism can be customised by providing *revivers* (for deserialisation) and *replacers* (for serialisation). Knowledge of specific revivers could still be used to initiate DoS attacks.

[14] In JRuby, the crucial behaviour showing how recursion is controlled can be found in `org.jruby.runtime.Helpers`, see `goo.gl/xc5mMK`

[15] `https://www.ruby-lang.org/en/news/2014/12/25/ruby-2-2-0-released/`

There are several alternative serialisation mechanisms outside the standard. This includes the *XMLSerializer* that is part of the Mozilla JavaScript extensions[16]. However, at the time of the study, this was not supported by any major web browser, including Firefox. *js-yaml* is a popular library that supports the YAML format[17]. However, map objects are currently not supported (in version 3.6.1) and attempts to serialise maps lead to a `YAMLException` being thrown.

JavaScript arrays (but neither objects nor maps) have a monotonic stringify method (`toString()`), but the study did not uncover a suitable trampoline to exploit this.

6.7 Conclusion

In this chapter, we have discussed the study of multiple YAML libraries covering ten languages for DoS vulnerabilities. In this study we have discovered seven previously unreported issues. The root of these vulnerabilities, YAML anchor/aliases is a major feature of the language which facilitates ease of hand-editing, readability and compactness. Some libraries do offer facilities to restrict the use of this feature with reference counting, depth limits and checks for circular references.

[16] `https://developer.mozilla.org/en-US/docs/Web/API/XMLSerializer`
[17] `https://github.com/nodeca/js-yaml`

Chapter 7

Detecting Injection Vulnerabilities

As discussed in Chapter 3, *arbitrary code execution* vulnerabilities are due to a program accepting untrusted input from deserialisation, and the input flowing to a security sensitive method.

It turns out that all of the known deserialisation vulnerabilities in Java exploit at least one dynamic feature. For instance, the Apache Commons Collections attack [69] exploits runtime reflection and the Groovy vulnerability (CVE-2015-3253 [52]) leverages Java's dynamic proxy feature. This suggests that a program where the deserialisation of an object leads to the invocation of such a feature is inherently vulnerable. For instance, if Java deserialisation (i.e., `readObject()`) triggers an execution of `Class::getMethod(..)` and `Method::invoke(..)`, and there is a flow from a field of an object in the deserialised object graph to the parameters of those methods, then this can be used easily to launch a code execution attack (e.g., via a reflective invocation of `Runtime::exec(..)` or a DoS attack (e.g., by allocating large data structures to exhaust memory, like `new ArrayList(size)`). We refer to the call sites of these features as *dynamic sinks*, because beyond these call sites it might be difficult or impossible for an analysis to reason, and the data may possibly flow to a security sensitive sink method such as `Method::invoke`.

To the best of our knowledge, there is only one tool for detecting serialisation vulnerabilities: (*Gadget Inspector*[1]) . This tool is purely static. It is based on call graph construction and data flow analysis to find potential serialisation vulnerabilities. It works by finding *gadget chains* - a call chain ending with an invocation of a security-sensitive method. Since the tool is static, it is affected by precision issues, and it also requires the user to manually examine and confirm the results (due to false positives).

[1] https://github.com/JackOfMostTrades/gadgetinspector

To improve on the deficiencies of a purely static analysis, we propose a hybrid analysis to detect potential Java serialisation vulnerabilities. The proposed method considers call sites that have reflective targets as unsafe dynamic sinks. The analysis is composed of two steps: the first step is to run a static analysis to detect call graph chains linking deserialisation call sites with dynamic sinks. We also look for data flows from deserialised objects to dynamic sinks via heap access paths.

In the second step, we use the results from the static analysis and a dynamic technique, fuzzing, to construct actual input objects. By construction, the deserialisation of those objects triggers the invocation of dynamic sinks with tainted input. This technique is unsound due to the dynamic step as it cannot generate all malicious inputs. However, experimental validation suggests that it is able to produce objects that can potentially be used for malicious purposes with deserialisation as an attack surface.

7.1 Running Example

A running example of a serialisation vulnerability is shown in Listing 7.1. To refresh some terminology, we refer to methods triggered by `readObject` as trampoline methods. `HashMap` in the Java collections library is a trampoline class for `Object::hashCode` and `Object::equals`. These methods are triggered in order to recompute the state of the `HashMap` from serialised data. Source objects are serialisable objects with trampoline methods. These objects are sources for data to enter the system, and there is a potential vulnerability if that data flows to a dynamic sink [15].

The `Container` class has two fields of types `Element` and `A` (`A` is an abstract class with a concrete implementation `C` and possibly other implementations (e.g. `B`) on the class path). When a stream containing a source object of type `Container` is deserialised using a `hashCode` trampoline on the `Container` object, it triggers the `getHash` method, depending on the type of the `property` field in the serialised object. If the object is of type `C` it invokes an arbitrary method specified in a string which is also obtained from the serialised object. If a program deserialises untrusted input, and if these classes are in the class path they can be utilised by an attacker to execute a method from an arbitrary class.

```
1 public class C extends A {
2   @Overrides
3   int getHash(Element e) {
4     return e.hashCode();
5   }
6 }
7
8 public class Container implements Serializable {
9   Element e;
10  A property;
11
```

```
12   public int hashCode() {
13     return property.getHash(e);
14   }
15 }
16
17 public class Element implements Serializable {
18   String methodName;
19   String className;
20
21   public Object valueOf() throws Exception {
22     Class klass = Class.forName(className);
23     Method m = klass.getMethod(methodName);
24     o = m.invoke(klass.newInstance());
25    }
26
27   public int hashCode() {
28     return valueOf().hashCode();
29   }
30 }
```

Listing 7.1: Java code with serialisation vulnerability.

7.2 Analysis

The analysis runs in two steps. In the first step, the static analysis takes a Java library as input, and performs on-the-fly call-graph construction and points-to analysis with trampoline methods as entry points to obtain information about the flow of values through the heap. We are interested in finding flow of values involved in a security violation (a flow of a tainted value to a dynamic sink during deserialisation). A heap access path has information about the objects and fields which participate in the flow of a value from a serialisable object to a parameter of a dynamic sink. In the second phase, an object is reconstructed based on these results. The trampoline method is executed with the object as an argument, and we verify whether the source-to-sink method execution path is observed. If it is not, we use the heap information as a seed for a fuzzer to explore inputs for a source-to-sink execution.

7.2.1 Static Analysis

In this section, we present the static analysis components of the specification. The static analysis uses a combination of points-to analysis and call graph construction to find potential security violations. We re-use the definitions from sec. **??** for the points-to graph and call graph. We also use the points-to graph for alias analysis. Two variable vertices v_1, v_2 *alias* if there is an object vertex $o \in O$ and paths consisting of $flowsTo$ edges from o to both variable vertices v_1 and v_2. Aliasing means that both variables can point to the same memory location.

Native Heap Allocations

Java deserialisation involves heap manipulation that does not use regular constructors for objects. The proposed analysis compensates for this by creating pseudo-objects. One approach [7] is to create pseudo-objects for public fields and formal parameters for all public methods, setting these public methods as entry points and then running the points-to analysis. If the points-to analysis is field-sensitive, this would result in a points-to graph with points-to sets for instance fields. Since deserialisation natively constructs objects from stream data, this approach does not suffice. It is possible to construct an object using stream data that might not be possibly to create using the public API for the object. Another issue with this approach is that the pointer analysis is expensive. A simpler approach is to create objects for *all* fields of a potentially deserialisable object. A pseudo-object's fields are set to recursively point to other pseudo-objects of their declared types. For instance, consider the statement `Object obj = ois.readObject()`. The points-to set for the variable `obj` will be a collection of pseudo-objects. In turn, the fields of these objects would point to pseudo-objects. These objects represent object creation due to deserialisation.

Definition 7.2.1 (Pseudo-objects)**.** In a points-to graph, we add a set of pseudo-objects, S, where

- $S \subseteq O$ is the set of vertices representing pseudo-objects arising from deserialisation in a program.
- $o \in S$, iff o is an instance of `Serializable`.

Heap Access Path

A points-to graph as defined above can be used to define a *heap access path* [97] between variables. A heap access path between v_1 and v_2 consists of a sequence of load edges where the destination (sink) of an edge in the sequence aliases with the source of the next edge, v_1 is the source of the first and v_2 the destination of the last edge in the sequence. This models (nested) field access in a programming language, i.e. statements like `foo.f.g`. By accounting for aliases, this also covers field access with intermediate variables, in programs like `x = foo.f; x.g;`.

Figure 7.1 shows the heap graph for the example Java code. The `String` object (line 24 in Listing 7.1) flows from the source object `Container` via its element field, labelled $\langle e \rangle$ in the graph and finally the field labelled by $\langle className \rangle$ in the *Element* instance. Hence, the heap access path is $Container \xrightarrow{e} Element \xrightarrow{className} String$.

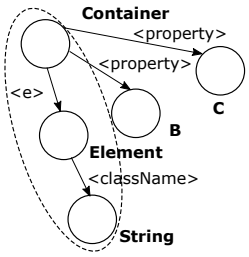


Figure 7.1: Heap graph.

Source Methods and Objects

We define source methods as methods triggered by deserialisation, and source objects as the receivers of these methods.

Definition 7.2.2 (Source Methods and Source Objects)**.** Given a call graph for a program, a method m is a source method if:

- it has a path in the call graph from *readObject* to m
- For the call site that targets m and that is reachable from *readObject*, the base variable (the first argument at the call site) points to a pseudo-object, which is defined as the source object.

Sink Methods and Objects

We define sink methods as security-sensitive reflective methods (hence, dynamic sinks), and sink objects as arguments to these methods.

Definition 7.2.3 (Sink Methods and Objects)**.** Given a program, the set of sink methods, $sinkMethods \subseteq M$ is defined where

- There is a path in the call graph from *readObject* to $sinkMethod$
- M is the set of methods in the program
- For call sites with sink methods as a target, the arguments to the method are defined as sink objects.

Security Violation

Security violations can be found statically by using the results of the points-to analysis and the resulting call graph. We define a security violation as follows.

Definition 7.2.4 (Security Violation)**.** Given a source object s with source method, m, we define a security violation as paths in the call graph and points-to graph with the following properties:

- there is a path in the call graph from a sink method m to a sink callsite, c
- a sink object that is reachable via a heap access path from the source object, and to which an argument of the callsite c, points-to.

In Listing 7.1, the source object is an instance of `Container` as it has a trampoline method, `hashCode`, in the call graph. The dynamic sinks are the call sites for `Class::forName`, `Class::getMethod` or `Method::invoke`. The field `className` aliases with a parameter for `Class::forName`, and since that field is in an access path from the source object, we consider the call site with the target `Class::forName` to be a dynamic sink as well as a tainted call site.

7.2.2 Dynamic Analysis

The dynamic phase uses the output of the static analysis, which are heap access paths to reconstruct an object with the same properties as a heap access path, using reflection. In the given example, the heap access path, $Container \xrightarrow{e} Element \xrightarrow{className} String$, consists of `Container`, `Element` and `String` objects. An edge in the path corresponds to the source of the edge being an object, and the target, a field of the object. After constructing the object we execute the trampoline method, which is `hashCode` in the example. Instrumentation is used to observe if we can trigger the call at the dynamic sink site. If this is not successful, we use fuzzing and attempt to observe the same effect.

Constructing Object from Heap Path

As listed in algorithm 1, the procedure takes a heap path as input. The end of the path is a variable pointing to a string, for which an object is insantiated and the procedure traverses the path in the reverse direction skipping assign edges. If it encounters a field or array load edge, it instantiates an object for the preceding variable and loads the current object into that field. It then sets that object as the current one and proceeds over the rest of the path. Finally, the source method is executed with the resulting object as a parameter.

Dynamic Taint Analysis

As introduced in Section 2.4.3, dynamic taint analysis can be used to monitor if a tainted object reaches a sink during the execution of the trampoline method. A dynamic taint analysis consists of predefined sources of taint, predefined sinks and propagator methods that transfer taint from one value to another through computations (e.g., string concatenation). Without propagator methods, only the values obtained at sources will be marked as tainted. Taint can be derived only through propagators.

Algorithm 1 Algorithm for Object Construction from Path

```
1:  function ConstructObject(path)
2:      object ← SelectObject(path.end)
3:      current ← path.end
4:      while current.edge_in ≠ null do                          ▷
5:          if Label(current.edge_in) ≠ assign then
6:              tmp ← SelectObject(current.prev)
7:              if IsArray(tmp) then          ▷ check if the object is an array
8:                  tmp[0] = object
9:              else
10:                 field = Label(current.edge_in)
11:                 tmp.field = object
12:             object = tmp
        return object
```

Definition 7.2.5 (Elements in Dynamic Taint Tracking)**.** We express the elements used in dynamic taint analysis as a triple, $\langle \mathcal{I}, \mathcal{S}, \mathcal{P} \rangle$

- Sources $\mathcal{I} = \{\langle m, t_i \rangle, ...\}$
- Sinks $\mathcal{S} = \{\langle m, t, h \rangle, ...\}$
- Propagators $\mathcal{P} = \{\langle m, i, j, t_k, t_l \rangle, ...\}$

t is a set of taint labels, h a set of methods that handle tainted input flowing to a sink, m is the set of methods in the program, i and j denote the parameter that is tainted and the variable that it propagates taint to, respectively.

Fuzzing

The object graph may require further elaboration for execution to reach the sink site due to guards in the code. For example, the object constructed from the heap access path will have a `Container` object with the `property` field pointing to `null`. Consequently, invoking the trampoline method will result in a null dereference before reaching the dynamic sink.

A technique to automatically generate random inputs that can explore different values for guards in the code is fuzzing [116]. Generation-based or mutation-based fuzzing [111] can be used to produce random serialised data. A mutation-based fuzzer can be seeded with serialised data, which it can then mutate randomly to generate new inputs. The drawback with this approach is that we are interested in valid serialised data that is syntactically valid as we are interested in semantic vulnerabilities [135]. Another approach is to use a generation-based fuzzer that generates input from a model or specification, such as the grammar for the binary serialisation format. It is

also possible to use random test input generation techniques (e.g., Randoop [133]) for Java to generate the objects.

In the case of the example heap graph in Figure 7.1, the edges that can be used to augment the heap access path are the two edges labelled ⟨*property*⟩ for the field property in the `Container` object. That is, the `property` field can point to either an object of type `C` or `B`. If the fuzzer generates an input with the `property` field set to type `B` it will not reach the dynamic sink. In another fuzzing run, if it picks `C`, execution will reach the dynamic sink. Once an input is generated, the trampoline method, `hashCode` is executed with root of the heap access path, the object of type `Container` as the receiver and if the sink, `Class::forName` is reached the fuzzer has found a serialisable object that can reach a dynamic sink on deserialisation.

7.3 Methodology

In order to evaluate this approach, we implemented the analysis in Doop [30], a datalog-based framework for specifying Java pointer-analyses. We adapted Doop's open-program analysis features to approximate Java deserialisation as described in the analysis section. The analysis pipeline consists of points-to computation using on-the-fly call graph construction for the target Java library, and computing heap access paths for security violations. In the evaluation, we used Randoop [133] for fuzzing. Randoop is a test-oriented fuzzing tool that takes a set of classes as input and generates a sequence of statements. These sequences are created incrementally. First a method/constructor is selected randomly from the classes given as input to Randoop. A statement is created using the selected method or constructor (a new statement or a method invocation). The arguments (if any) to the invocation are selected from the output of previously constructed sequences or random values (for primitives and strings). When a new sequence is created, it is executed and checked against a set of contracts to output tests. We modified Randoop so that, upon the creation of a new sequence, it selects a variable from the method sequence and passes it to a checker. The checker invokes trampoline methods on the variable and if the execution reaches a sink method, it is classified as a failing test and output on completion of sequence generation. Randoop was provided the set of classes on the heap access path as input.

Experimental Setup. The experiments were performed on an Intel(R) Core(TM) i7-8700 CPU @ 3.20GHz with 64GB of RAM on Linux Ubuntu 18.04.3. Doop[2] was run using the Java 8 platform as implemented in Oracle's version 8 of the JDK (build 1.8.0_221-b11). Doop was run with the options for a context-insensitive analysis with support for light reflection.

[2]Doop version built from commit: `https://bitbucket.org/yanniss/doop/commits/ba731a63c90e94f9e94afca39ff15c5082bf868d`

Table 7.1: Overview of experiments

Library	Time (s)	Sinks	Paths	Objects
bsh-2.0b5	655	1	1	0
clojure-1.8.0	timeout	n/a	n/a	n/a
commons-beanutils-1.9.2	786	0	0	0
commons-collections-3.1	1611	1	1	1
commons-collections4-4.0	681	1	1	1
groovy-2.3.9	5204	3	3	0
hibernate	3397	2	3	0
jython-standalone-2.5.2	timeout	n/a	n/a	n/a
rome-1.0	390	1	0	0

Dataset. We used a number of libraries obtained from Frohoff's *ysoserial*[3] repository. As we specifically focus on reflection-enabled vulnerabilities, we excluded libraries with DoS vulnerabilities. We also chose libraries with vulnerabilities for `hashCode` and `compare` trampolines, since these are the most common. For dynamic sinks, we have considered methods that load classes, and reflective calls to classes to fetch their method(s).

7.4 Results

In evaluating the approach, the analysis reported two of the reflection-based injection attacks from the *ysoserial* collection of vulnerabilities. Table 7.1 shows an overview of the results from the static analysis: the running time, number of dynamic sinks detected, and heap access paths from the sink to the source object with the trampoline method.

For the Apache Commons Collections (ACC) libraries, we were able to reproduce an object that, when used as a parameter for the source method, gives rise to an execution path from the trampoline to the dynamic sink with a tainted flow. We were able to produce objects for two variants of payloads for the library. For the vulnerabilities with the `InvokerTransformer` class, we were able to reproduce an execution path to a dynamic sink (`getMethod`) from `hashCode` and `compare` trampolines, respectively. Of these, only one required further fuzzing and for the other, we used reflection to simply instantiate the objects on the heap access paths, resulting in an object that produced the execution path on invoking the trampoline method. Randoop was able to produce an input with the aid of the heap access paths in under 2 minutes. Without this aid, it produced no results within a time limit of 30 minutes. In comparison, *Gadget Inspector*, the only other known analyser, produced four call chains for the ACC vulnerabilities, and in two cases, it missed the full set of gadget classes in the actual exploit. A benefit

[3] https://github.com/frohoff/ysoserial

of the proposed analysis is that it produces executable code to generate the input that reaches the target sink, which aids in the comprehension of the analysis reports.

We also used an alternative fuzzer AFL [197] - a security-oriented greybox fuzzer that uses instrumentation for coverage guided generation of inputs. As AFL targets C programs, we used the Java front-end for AFL Kelinci [98]. In contrast to Randoop, this tool uses reflection to construct the object during fuzzing. The heap access paths obtained for the same libraries were augmented with additional points-to facts from the heap graph and the trampoline method executed for vulnerabilities. Using fuzzing to produce a successful execution from the trampoline to the sink site required only under three minutes. But using fuzzing without the heap access path as a guide for the fuzzer produced no successful results after ten minutes of fuzzing.

We also implemented a grammar-based fuzzer for the Java serialisation protocol to confirm similar results. Unlike the Randoop-based fuzzer, which generates code to construct the object, the grammar-based fuzzer generates object graphs.

7.5 Conclusion

This chapter presents a hybrid analysis to detect serialisation vulnerabilities. We evaluated the proposed approach on a dataset of real-world serialisation vulnerabilities. The results of evaluating the analysis on the *ysoserial* vulnerabilities dataset, which shows two confirmed reports, are promising. The analysis uses a static pre-analysis to improve the efficiency of fuzzing. The evaluation shows that without the results from the pre-analysis, Randoop was not able to produce a result within a time limit of 30 minutes.

On the other hand, this approach can also be considered a dynamic post-analysis that improves the precision of a primarily static analysis. The only alternative method available, *Gadget Inspector*[85], may suffer precision issues due to missed branch conditions, and it requires manual inspection of the reports. The dynamic post-analysis addresses this by selecting true positives, thereby removing all false positives. This may reduce the recall of the analysis by missing true positives. However, as mentioned earlier, the analysis is already unsound, like most non-trivial program analyses.

Chapter 8

Mitigation

In this chapter we discuss strategies to mitigate the security issues that arise from deserialisation vulnerabilities. We provide a classification of the various methods around prevention at buildtime and detection during runtime. We also present an approach to prevent DoS vulnerabilities that arise from deserialisation and discuss relevant techniques against injection attacks.

8.1 Strategies Against Untrusted Deserialisation

Most of the current mitigation strategies are based on defence-in-depth approaches, that is, at the outermost level, the network perimeter is monitored for serialised objects. At the next level, instrumentation is used to monitor serialisation, or the `ObjectInputStream` can be wrapped to perform preliminary checks before its functionality is used. Subclassing `ObjectInputStream` can implement *inclusion lists* or *exclusion lists* of deserialisable classes[1]. For applications that use third-party libraries that utilise serialisation, instrumentation based approaches are feasible to guard against open deserialisation. For example, NotSoSerial[2] monitors calls to `resolveClass` in the `ObjectInputStream` and prevents deserialisation of objects that are not in the whitelist. The subclass inspired approach has already been implemented in `ValidatingObjectInputStream` in the Apache Commons IO library and a filter-based stream is available in JEP290 for Java 9 [128]. Neither of these are completely effective [119] as deserialisation of system classes (as we describe) can result in DoS attacks.

We consider countermeasures along three dimensions: object structure, polymorphic deserialisation, encapsulation and security sensitive methods. For Java serialisation, a filter mechanism described in JEP 290 [128] is available that can be configured

[1] http://www.ibm.com/developerworks/library/se-lookahead
[2] https://github.com/kantega/notsoserial

to impose restrictions on the object structure: depth of input graph, total count of internal references and the input size. For most other serialisers, there are no restrictions on parsing other than disabling the alias feature and custom type tags. Type filters on the stream and lookahead deserialisation that works with type exclusion/inclusion lists can limit the types that are instantiated during deserialisation. However, this may not be sufficient protection against DoS attacks using collection types which are commonly used and are likely to be included in whitelists.

Extralinguistic object reconstruction breaks the safety guarantees provided by encapsulation and rich type systems such as confinement/ownership types. It is possible to verify the stream to check if these properties are still satisfied (e.g. check if internal references leak outside an object's scope). The security manager and static/dynamic taint checking are some measures against the invocation of sensitive methods with unsanitised arguments as inputs.

Restrictions on Classes to Deserialise

The main defense against remote code execution attacks that are based on deserialisation is to allow only certain classes to be deserialised. This strategy is employed in JEP290 [128], a filter mechanism for Java native deserialisation. Filters can employ inclusion/exclusion lists to control which classes can or cannot be deserialised when an object is read from a stream. This strategy is effective against known attacks but it cannot block new exploits.

Restrictions on Graph Size and Topology

A strategy against DoS attacks resulting from deserialisation is to **restrict the size or topology of graphs** created during parsing. Restrictions can be on the depth from a root node, the number of children per node, the overall vertex or edge count, the kind of node, or any combination of those. This strategy is utilised in JEP290 to control the input of streams of serialised Java objects. The parameters that can be controlled include depth (`maxdepth`), edge count (`maxrefs`), and type restrictions. Array sizes and the overall data size of the stream can also be restricted. Similarly, the JAXP XML parser API [1] uses processing limits for parsing XML, and the JSON standard suggests that parsers set limits, including on the maximum depth of nesting [94, Sect. 9].

However, this is not a guarantee against exponential blowup, as it does not consider how a tree or graph can be expanded by the client program even though references may not be supported natively in the deserialisation format. For instance, some data formats do not support graphs, i.e., they do not allow for object identifiers / reference types and the data is serialised as a tree. Notable examples are Protocol Buffers and JSON, which

natively do not have support for references. Hence, it might not be possible for a parser to immediately produce a composite object graph with the required type. Serialisation that relies on tree-based formats, such as XML or JSON, rely on encoding object references in the tree. The Java serialisation library XStream uses XPath or object ids, XMLEncoder uses XML ID and IDREF attributes and BSON uses ObjectId's for object reuse in the tree representation.

Even though file format specifications may suggest that it is not possible to construct an object graph susceptible to billion laughs type of vulnerabilities, some parsers may relax the requirements, and some applications may convert the parsed tree into a many-to-many composite. An example is that the PDF specification states that a page tree node cannot have multiple parents. However, as discussed in Chapter 5, some parsers process ill-formed page trees in the document structure. XML and SVG formats are known for the security issues they pose [171], and there has been prior work by Heiderich et al. [86] on sanitising SVG files with the *SVGPurify* tool. *SVGPurify* removes malicious content (e.g., SVG-based JavaScript) from an SVG file. However, the tool does not cover DoS attacks using malicious SVG files.

We make a case that sanitisation must happen prior to the semantics phase (e.g. rendering an SVG or converting a YAML parse tree to native data structures). It is common for the processing of the tree to be implemented using visitors. Visitors normally do not track the depth and processing deep syntax trees can result in stack overflows. For example, the JVM specification [129] limits array dimensions to 255 yet the Java language specification has no limits, and a parser would fail in processing an array variable with a large number of dimensions if it exhausts the stack. The same holds for the YAML and SVG cases, a parser has no issues parsing the input (underlying XML parser produces a parsed document, low-level YAML parser produces the YAML node structure). librsvg handles the error during processing by keeping count of the instantiated objects created via the use tag, which can be expensive.

These restrictions are easy to implement and impose low runtime overhead. However, their soundness (ability to detect all malicious input in the context of a particular program) and precision (ability to flag only malicious input) is highly dependent on the correct calibration, i.e. the setting of thresholds for the respective parameters by engineers who deploy and configure the application.

Resource and Performance Quota, Sandboxing

A popular approach is to sandbox the execution of critical methods, for instance, by using containerisation. Here, a runtime environment is configured with resource (CPU, time, memory) quota. If resources are exhausted, the application is notified and the impact on the application is controlled. Many language runtimes offer similar facilities.

For instance, when a Java application encounters stack overflow or out of memory errors, the impact is limited to a particular thread, and application servers can gracefully handle this situation.

This is not sufficient to prevent DoS attacks, as the restarting of threads is computationally expensive, and DoS attacks can still be constructed by creating error logs exhausting disk space. Moreover, the crash reports created are not necessarily revealing the security nature of those events, which may appear as programming or configuration problems. The resource contracts proposed in [58] makes potential resource exhaustion transparent by enforcing resource usage post conditions in method calls, and raising proper security exception to communicate suspicious resource consumption. There is some overhead to this method though as it requires instrumentation of application code. As before, soundness and precision are highly dependent on calibration.

An example of using sandboxing against deserialisation remote code execution attacks is the method proposed by Cristalli et al. [46]. They detect malicious executions by comparing them against trusted code executions that are triggered by deserialisation. This information is collected in a phase that first exercises the application, and if deserialisation, in production, triggers suspicious execution path, it is blocked.

Performance-directed Fuzzing

Fuzzing seeks to generate input to reveal unexpected behaviour, usually directed by a fitness function that evaluates input and prioritises highly ranked input for future input generation. While many fuzzers uses code coverage or similar metrics, performance characteristics such as declining performance can be used in fitness functions. This can then be used to reveal input that exhausts resources, for instance, by triggering worst case complexity behaviour of implemented algorithms. Assuming that the resource constraints used in the fuzzing campaign are comparable with the runtime constraints, this approach is precise, i.e., it does not produce false positives. However, it is not sound, as there is no guarantee that a fuzzing campaign exercises all possible execution paths with all possible inputs. Examples of this approach are PerfFuzz [102] , SlowFuzz [142] and HotFuzz [25].

Static Analysis

We studied preventative approaches in earlier chapters. We also used the test set of programs from Chapter 5 as input to existing static code analysis tools that can detect security vulnerabilities. We evaluated the test set on light-weight static analysis tools

Table 8.1: Comparison of mitigation strategies

Method	**Sound**	**Precise**	**Lifecycle**
Restrictions on graph size and topology	calibration-dependent		Runtime
Resource and performance quota, sandboxing	calibration-dependent		Runtime
Performance-directed fuzzing	no	yes	Buildtime
Static analysis	soundy	no	Buildtime
Hybrid analysis	no	yes	Buildtime

such as SonarQube[3], lgtm[4] and SpotBugs[5]. SpotBugs did not find issues related to recursion in the test dataset. The standard bug patterns that are related to the issue are limited to "Collections should not contain themselves" and "An apparent infinite recursive loop". Similarly, lgtm and SonarQube did not report the vulnerabilities.

Hybrid Analysis

Hybrid program analyses try to blend static and dynamic techniques in order to combine their respective advantages. They have shown promise in program analysis, in particular in order to address soundness issues in pure static analyses [27, 82]. To the best of our knowledge, there is no work to use a hybrid analysis to detect the kind of DoS vulnerabilities we are interested in. The work we present in [153] uses a hybrid analysis to detect code execution attacks in Java's serialisation protocol. In particular, fuzzing is used to identify true positives in the results returned by the static analysis part of the analysis.

Summary

Table 8.1 summarises available countermeasures and prevention strategies. We have discussed the respective strategies in detail and in the following sections we propose improved approaches for detecting for DoS vulnerabilities and injection vulnerabilities in serialisation.

[3] `https://www.sonarqube.org/`
[4] `https://lgtm.com/`
[5] `https://spotbugs.github.io/`

8.2 Runtime Filter for Deserialisation and Parsing

Countermeasures for XML Entity Expansion attacks and quadratic blowup based on prohibiting entities or limiting the depth of nested entities are excessively restrictive. For Java serialisation, a filter mechanism described in JEP 290 [128] is available that can be configured to impose restrictions on the depth of the input graph, the total count of internal references and the input size. For the YAML parsers we studied, there are no restrictions on parsing other than disabling the alias feature, recursive data structures for the Ruby YAML library, and disabling type tags for the Python, Ruby and Java YAML libraries.

The approach we propose is a more granular filter that is more permissive than the available techniques. We attempt to estimate the size of the call tree and prevent further processing (conversion to native data structures) after parsing the YAML representation graph if the properties of the object graph suggests a call tree with a non-viable execution time. This technique will result in an analysis that rejects fewer object graphs as malicious, i.e., it returns less false positives.

An approach such as rejecting inputs with references, reference counting or limits on graph depth rejects innocuous object graphs. To reject inputs with references is a very imprecise approach. A restriction on object graph depth is also not optimal as there are cases where deeply nested graphs are benign inputs. One example being serialised linked lists. In addition, the test vectors that caused DoS in the experiments are not excessively deep. Reference counting thresholds are of little help as we have demonstrated that test vectors where the resulting graph has as few as 98 internal references can cause resource exhaustion. With regard to setting a threshold, one approach is to set a threshold that is guaranteed to execute in a few seconds, which will let in anything but malicious inputs. In this case there is no need to finely tune the threshold to a particular execution environment.

In a call tree, a call chain is a sequence of edges, $e_1, ..., e_n$, where the target of e_i is the same as the source of e_{i+1} in the call tree. For *SerialDoS*, each call chain in the call tree is a path from the root of the graph to a leaf. The number of call chains executed is the number of *root–leaf* paths in the object graph. A filter can thus be setup to reject object graphs that can result in call trees that have call chain counts above a specified threshold.

Algorithms for Path Counting

For a directed acyclic graph, $G(V, E)$ where V and E are the sets of vertices and edges respectively, the number of simple paths between two vertices can be counted using either memoisation and depth-first search with a runtime of $O(|E| + |V|)$, or by topologically sorting the graph and adding the number of paths arriving to each vertex

from its predecessors [172]. In the memoisation case, the number of paths is given by the recurrence equation:

$$\text{Paths}(s,d) = \begin{cases} \text{if } s = d & = 1 \\ \text{otherwise} & = \sum_{v \in Successors(s)} \text{Paths}(v,d) \end{cases}$$

We implemented a proof of concept in C++ (source code available in the repository[6]) using the LibYAML parser. It first converts a YAML document to a directed acyclic graph that captures objects (lists, anchors and mappings) as vertices and containment of any of these objects, either as an alias or a concrete object within another, as an edge. In the implementation, the number of paths from the root to the leaves is computed using depth-first search and memoisation to avoid exponential runtime. If the number of paths exceed a user-defined threshold it can be rejected as unsafe to deserialise. We tested the prototype on the test vectors with nested list structures, on hardware with the same configuration as that in the previous experiments. It checks the inputs in an average of 27ms. Therefore, this technique could be used as an effective precondition check for parsers to reject potentially malicious input. In fact this technique is more efficient than the approach used in the fixes for librsvg and svg-sanitizer following our vulnerability reports.

8.3 Dynamic Type Checking

One of the issues with the existing filter mechanism for Java binary deserialisation (JEP290) is that it is limited to checking types that appear in the serialised data stream. This is a check against a whitelist or blacklist of types that are either allowed or disallowed from being deserialised by the application. This can be configured at the deserialisation call site in the program, or globally for the application in the environment. Another issue is polymorphic deserialisation and generic types in Java.

Generics allows classes to be parameterised with different element types (e.g., the class parameterised with the type `T` as `List<T>` can take a `String` as a type parameter resulting in `List<String>`). To see how this affects serialisation, consider the code in Listing 8.1. The object, `hm`, is of type `HashMap<K, V>` with type parameters, `String` for both its keys and values. Due to how Java generics is implemented these types are erased and replaced with the bounds from the class declaration (in this case, `Object`). The type parameters are not reflected in the serialised data, which has the concrete types of the objects (in the example, `String`). There is no facility for type checking

[6] `https://bitbucket.org/unshorn/dosyamlpub/src/master/src/filter/`

during deserialisation, and the mechanism returns an `Object`. The issue is that the current mechanism in JEP-290 does not check whether or not the data is well-formed with respect to the type that is intended by the programmer. Fields with type parameters in their declarations are deserialised polymorphically (i.e., if the value for the object on the stream checks correctly against the field's declared type). We propose a method to type check the data on the stream against a type specified at the deserialisation call site.

```
1 HashMap<String, String>  hm = new HashMap<>();
2 hm.put("foo", "foo");
3 FileOutputStream fos = new FileOutputStream("testfile");
4 ObjectOutputStream oos = new ObjectOutputStream(fos);
5 oos.writeObject(hm);
6 oos.close();
7 FileInputStream fis = new FileInputStream("testfile");
8 ObjectInputStream ois = new ObjectInputStream(fis);
9 obj = ois.readObject();
10 hm = (HashMap<String, String>) obj;
11 ois.close();
```

Listing 8.1: Polymorphic Deserialisation

8.3.1 Specification

First we define what safe deserialisation is with respect to a type supplied by the programmer and any given input stream. The definition ensures that the runtime type of the object graph in the stream is type-compatible with the type specified by the programmer.

Definition 8.3.1 (Safe deserialisation)**.** Given a serialised stream, S, we consider the object, o, read from the stream with the type $C\langle\theta_1, ..., \theta_n\rangle$ and a type specified by the programmer, $T\langle\theta_1, ..., \theta_m\rangle$. Deserialisation is safe if the type C subsumes T w.r.t. the relation $\leq$.

- The type $C\langle\theta_1, ..., \theta_n\rangle$ is inferred from the data in the stream.
- The $\leq$ relation is the subtyping relation. It is the reflexive, transitive closure of the direct subtype relation between Java classes. If $C \leq T$ and for all type parameters $\theta_{iC} \leq \theta_{iT}$, then deserialisation is safe.

We implemented a proof of concept for safe deserialisation using dynamic type checking as specified above. The prototype contains two elements: a parser that builds a representation of an input stream and a type checker. If the type check fails it throws a security exception. The user must specify a type and provide a stream as an input. The parser does not deserialise the stream but processes it as a sequence of bytes. A tree representation is built from the sequence of bytes, which is the object graph

annotated with type information. Note that generics is not persisted to the stream. Normal serialisation proceeds only if the type check is successful.

The benefits of this technique is that it gives the programmer fine-grained control over polymorphic deserialisation and over the structure of the object graph before it is deserialised.

8.4 Conclusion

In this chapter, we have discussed a variety of prevention and protection mechanisms that are available against DoS attacks and injection vulnerabilities due to deserialisation. These mechanisms can be costly to deploy, and as an alternative, we have proposed a light-weight filter that makes deserialisation safer while preserving the ability to use features such as aliases and deeply-nested structures. We have also presented a more precise type checking mechanism for Java binary serialisation that is an improvement over simpler filters.

Chapter 9

Conclusion

In this thesis, we have we have pursued aspects of addressing security issues in Java serialisation. Attackers can exploit serialisation vulnerabilities to carry out code injection or DoS attacks. To detect and mitigate such attacks we have explored multiple program analyses. We then proposed a hybrid analysis that used the output of static analyses to guide the search for vulnerabilities. We have formalised a class of DoS vulnerabilities and presented effective mitigation measures against DoS attacks based on these vulnerabilities. We also conducted a systematic study of DoS vulnerabilities in parsers for YAML formats across numerous programming languages. We demonstrated how our analyses and other studies uncovered actual vulnerabilities in real world software.

In this chapter, we summarise the main contributions of the thesis, list previously unknown security issues that we discovered, and suggest directions for future work, and thus conclude this research.

9.1 Main Contributions

The main contribution of this research is the use of program analysis to detect security vulnerabilities in serialisation, and mitigate against potential attacks. One of the general principles from the research is how the model we proposed in Chapter 5 can be a useful abstraction for reasoning about this class of DoS vulnerabilities. Another principle is the importance of validating data using constraints other than those that are widely in use (e.g., depth of graph). Another observation is that limiting input grammars (e.g. in the case of YAML, SVG) are not available as features in most parsers and users of these libraries should be cautious when deploying them in environments where they are susceptible to processing untrusted input. More specifically, the contributions are:

- In Chapter 3 we studied serialisation features in Java and discussed how these features lend to weaknesses resulting in vulnerabilities. The technique to mitigate against attacks that is available within the language is a type and graph filter. Blacklisting classes from deserialisation is not effective as Java applications are open to extension due to dynamic linking. Whitelists are not effective as they require assurance that a class is not vulnerable. This assurance is normally obtained by manual inspection of the code. These are limitations in the builtin filter mechanism. We also demonstrated the limitations of existing static analyses to discover vulnerabilities in Java. Particularly, how their handling of serialisation is not sound, and how instances of missed alarms can arise.

- Our second contribution is a hybrid analysis to detect injection vulnerabilities in Java deserialisation. In our approach we combined both static and dynamic taint analyses to confirm potential reflection injection vulnerabilities in Java libraries that may arise during deserialisation. We showed how this technique is effective at addressing the false alarm issue of static analyses. Our approach is also beneficial in that it generates an executable witness for the alarm that programmers can further examine.

- Our third contribution is a static analysis to detect DoS vulnerabilities in parsing composite data structures. Data serialisation languages such as YAML and XML are covered in this study. We presented a formalisation of a class of DoS vulnerability and implemented an analysis for Java using Doop (a datalog-based static analysis framework). We showed the effectiveness of our analysis by uncovering multiple vulnerabilities in widely used Java libraries. We also demonstrated a process for re-using the exploits that were constructed as a result of our analysis to confirm vulnerabilities for the same format in libraries written in other languages such as PostScript, Rust and PHP. Evidently, this shows the usefulness of the formalisation and how it can effectively generalise to other languages.

- In Chapter 8 we discuss the mitigation techniques available to detect these vulnerabilities and we introduce novel techniques to sanitise data for data serialisation languages, binary Java serialisation and file formats such as SVG. We demonstrate the efficiency of our technique in comparison to other sanitisation techniques. We also show that it is more effective than available graph filter mechanisms.

- In Chapter 6 we study vulnerabilities in YAML parsers across various languages. We used the patterns identified in Chapter 5 to construct payloads and exercised the libraries with them. We studied their effects and in two cases, constructed custom payloads for JavaScript and PHP YAML parsers. As a consequence, we discovered seven previously unknown vulnerabilities.

9.2 Newly Discovered Security Vulnerabilities

Following the guidelines for responsible disclosure, we have reported all vulnerabilities to the libraries' developers/maintainers. We provide a timeline and the statuses of each of these vulnerabilities below.

Table 9.1: Status of reported bugs and vulnerabilities status.

Library	**Ver.**	**Status** (CVE/ Bug/Dup./ Pending	**Fixed date**
PDFBox	2.0.12	CVE-2018-11797	6-Oct-18
PDFxStream	3.6.0	CVE-2019-17063	27-Feb-19
PDFTk	2.02	Pending	
GhostScript	9.25	CVE-2018-19478	
Svg-sanitizer	0.13.0	CVE requested (pending)	20-Jan-20
Batik	1.11	Wont fix	-
Firefox	69	Duplicate	-
Drupal	6.x-8.x	-	25-June-20
Snakeyaml	1.23	Wont fix	
Qtsvg	5.14.1	Bug	29-Feb-20
Librvg	2.46.2	CVE-2019-20446	15-Oct-19
PHP PECL Yaml	2.0.4	Security bug	6-May-20
js-yaml	3.13.0	npm advisory 788	21-Mar-19

9.3 Concluding Remarks

Serialisation is a popular feature in all programming languages. Its use is common in web applications, distributed computing and configuration / data persistence. It is expected that newer languages will add support for general serialisation features in the form they are available for mainstream languages such as Python, Java (e.g., ability to serialise any data structure). Especially as such demand grows in domains such as distributed programming.

This is not the first work to explore taint analysis, open program analysis and fuzzing for security. However, to the best of our knowledge, it is the first to explore the combination of these approaches to serialisation security.

We proposed a formalisation for an analysis to detect a class of DoS vulnerabilities, we implemented the analysis using Datalog, and demonstrated how it can be effectively used to discover previously unknown security issues in popular Java software

(e.g. Apache PDFBox, SnakeYAML, Apache Batik). We have also used the results of analyses to construct exploits that demonstrated security bugs in software written in other languages (librsvg in Rust, GhostScript in PostScript... etc). We surveyed parsers for the YAML data serialisation language and showed that this class of vulnerabilities is common and that mitigation features are lacking. We also evaluated a hybrid analysis to detect injection vulnerabilities in serialisation. Finally, we proposed an effective approach to counter attacks based on these vulnerabilities using type and graph-based sanitisation of input as mitigation strategies.

9.4 Future Work

Possible directions for future work can focus on recall and precision of the analysers, and also adapting the techniques to function for additional serialisation technologies. This could be other serialisation formats or data serialisation languages and programming languages other than Java.

The static taint analysis that we used in Chapter 5 is based on pointer-analysis. An alternative approach is to use an IDFS-based interprocedural data flow analysis for detecting serialisation injection. An example is the FlowDroid [16] taint analysis tool for detecting information-flow vulnerabilities. As discussed in Chapter 3, there are limitations when this tool is used for serialisation-based taint analysis.

Currently, the analysis for detecting DoS vulnerabilities in Java libraries is purely static. Static analysers over-approximate, which results in false alarms for the patterns that it reports. A possible approach is to use micro-fuzzing [25] to fuzz the recursive functions reported by the static analysis. This can improve the precision of the results from the static analysis for DoS whilst at the same time reducing fuzzing time by only focussing on the reported functions rather than all recursive functions in the library.

Constraint-based approaches can be used to more precisely identify the topology and traversal patterns that our static analysis reports. For instance, symbolic execution can be used to check if the construction of topologies do not contain guards that restrict against the construction of vulnerable cross-referencing composite structures. Such path constraints can also be used to refine if unchecked recursive calls are possible with the inputs. We did investigate using Symbolic Java PathFinder for this purpose. However, it does not support subtypes for symbolic heap values in symbolic execution to enhance the precision of the static analysis for DoS. Support for this feature is something that can be studied.

The fuzzer that we use in Chapter 7 is purely random. It would be interesting to evaluate other guidance functions for the fuzzer, for instance path coverage in the library.

Bibliography

[1] Java api for xml processing (jaxp) – processing limits. `https://docs.oracle.com/javase/tutorial/jaxp/limits/limits.html`. [Online; accessed 9-September-2020].

[2] Jackson. `https://github.com/codehaus/jackson`, 2016. URL `https://github.com/codehaus/jackson`. [Online; accessed 31-October-2016].

[3] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, 1995. ISBN 0-201-53771-0.

[4] Mehmud Abliz. Internet denial of service attacks and defense mechanisms. *University of Pittsburgh, Department of Computer Science, Technical Report*, 2011.

[5] alech. Efficient denial of service attacks on web application platforms. `https://perma.cc/Y55U-3PLR`, 2011. [Online; accessed 11-March-2019].

[6] Nicholas Allen, Padmanabhan Krishnan, and Bernhard Scholz. Combining type-analysis with points-to analysis for analyzing java library source-code. In *Proceedings of the 4th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis*, SOAP 2015, pages 13–18, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3585-0. doi: 10.1145/2771284.2771287. URL `http://doi.acm.org/10.1145/2771284.2771287`.

[7] Nicholas Allen, Padmanabhan Krishnan, and Bernhard Scholz. Combining type-analysis with points-to analysis for analyzing java library source-code. In *Proceedings of the 4th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis*, SOAP 2015, pages 13–18, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3585-0. doi: 10.1145/2771284.2771287. URL `http://doi.acm.org/10.1145/2771284.2771287`.

[8] Saswat Anand, Corina S Păsăreanu, and Willem Visser. JPF–SE: A Symbolic Execution extension to Java Pathfinder. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 134–138. Springer, 2007.

[9] Saswat Anand, Edmund K. Burke, Tsong Yueh Chen, John Clark, Myra B. Cohen, Wolfgang Grieskamp, Mark Harman, Mary Jean Harrold, and Phil Mcminn. An orchestrated survey of methodologies for automated software test case generation. *J. Syst. Softw.*, 86(8):1978–2001, August 2013. ISSN 0164-1212. doi: 10.1016/j.jss.2013.02.061.

[10] James H Andrews and University of Western Ontario. Dept. of Computer Science. *Randomized unit testing: Tool support and best practices*. Citeseer, 2006.

[11] Tony Antoniadis, Konstantinos Triantafyllou, and Yannis Smaragdakis. Porting doop to soufflé: A tale of inter-engine portability for datalog-based analyses. In *Proceedings of the 6th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis*, SOAP 2017, page 2530, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450350723. doi: 10.1145/3088515.3088522. URL `https://doi.org/10.1145/3088515.3088522`.

[12] Andrea Arcuri and Lionel Briand. Adaptive random testing: An illusion of effectiveness? In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ISSTA '11, pages 265–275, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0562-4. doi: 10.1145/2001420.2001452. URL `http://doi.acm.org/10.1145/2001420.2001452`.

[13] Molham Aref, Balder ten Cate, Todd J. Green, Benny Kimelfeld, Dan Olteanu, Emir Pasalic, Todd L. Veldhuizen, and Geoffrey Washburn. Design and Implementation of the LogicBlox System. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. ACM, 2015. ISBN 978-1-4503-2758-9. doi: 10.1145/2723372.2742796. URL `http://doi.acm.org/10.1145/2723372.2742796`.

[14] Shay Artzi, Michael D Ernst, Adam Kiezun, Carlos Pacheco, and Jeff H Perkins. Finding the needles in the haystack: Generating legal test inputs for object-oriented programs. 2006.

[15] Steven Arzt, Siegfried Rasthofer, and Eric Bodden. Susi: A tool for the fully automated classification and categorization of android sources and sinks. 2013.

[16] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *Acm Sigplan Notices*, 49(6):259–269, 2014.

[17] Jean-Philippe Aumasson and Daniel J Bernstein. SipHash: a fast short-input PRF. 2012.

[18] aws. Working with AWS CloudFormation Templates. `https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/template-guide.html`, 2019. [Online; accessed 11-March-2019].

[19] Domagoj Babić, Lorenzo Martignoni, Stephen McCamant, and Dawn Song. Statically-directed dynamic automated test generation. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ISSTA '11, pages 12–22, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0562-4. doi: 10.1145/2001420.2001423. URL `http://doi.acm.org/10.1145/2001420.2001423`.

[20] David F. Bacon and Peter F. Sweeney. Fast static analysis of c++ virtual function calls. *SIGPLAN Not.*, 31(10):324–341, October 1996. ISSN 0362-1340. doi: 10.1145/236338.236371. URL `http://doi.acm.org/10.1145/236338.236371`.

[21] Victor R. Basili. The experimental paradigm in software engineering. In H. Dieter Rombach, Victor R. Basili, and Richard W. Selby, editors, *Experimental Software Engineering Issues: Critical Assessment and Future Directions*, pages 1–12, Berlin, Heidelberg, 1993. Springer Berlin Heidelberg. ISBN 978-3-540-47903-1.

[22] Jason Bau, Elie Bursztein, Divij Gupta, and John Mitchell. State of the art: Automated black-box web application vulnerability testing. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP '10, pages 332–345, Washington, DC, USA, 2010. IEEE Computer Society. ISBN 978-0-7695-4035-1. doi: 10.1109/SP.2010.27. URL `http://dx.doi.org/10.1109/SP.2010.27`.

[23] Jonathan Bell and Gail Kaiser. Dynamic taint tracking for java with phosphor (demo). In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ISSTA 2015, page 409413, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450336208. doi: 10.1145/2771783.2784768. URL `https://doi.org/10.1145/2771783.2784768`.

[24] Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. A few billion lines of code later: Using static analysis to find bugs in the real world. *Commun. ACM*, 53(2):6675, February 2010. ISSN 0001-0782. doi: 10.1145/1646353.1646374. URL `https://doi.org/10.1145/1646353.1646374`.

[25] William Blair, Andrea Mambretti, Sajjad Arshad, Michael Weissbacher, William Robertson, Engin Kirda, and Manuel Egele. Hotfuzz: Discovering algorithmic denial-of-service vulnerabilities through guided micro-fuzzing. *Proceedings 2020 Network and Distributed System Security Symposium*, 2020. doi: 10.14722/ndss.2020.24415. URL `http://dx.doi.org/10.14722/ndss.2020.24415`.

[26] Joshua Bloch. *Effective Java (2nd Edition) (The Java Series)*. Prentice Hall PTR, NJ, USA, 2 edition, 2008. ISBN 0321356683, 9780321356680.

[27] Eric Bodden, Andreas Sewe, Jan Sinschek, Hela Oueslati, and Mira Mezini. Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 241–250. ACM, 2011.

[28] Sergey Bratus, Michael E Locasto, Meredith L Patterson, Len Sassaman, and Anna Shubina. Exploit programming: From buffer overflows to weird machines and theory of computation. *USENIX; login*, 36(6):13–21, 2011.

[29] Martin Bravenboer and Yannis Smaragdakis. Strictly declarative specification of sophisticated points-to analyses. *SIGPLAN Not.*, 44(10):243–262, October 2009. ISSN 0362-1340. doi: 10.1145/1639949.1640108. URL `http://doi.acm.org/10.1145/1639949.1640108`.

[30] Martin Bravenboer and Yannis Smaragdakis. Strictly declarative specification of sophisticated points-to analyses. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '09, pages 243–262, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-766-0. doi: 10.1145/1640089.1640108. URL `http://doi.acm.org/10.1145/1640089.1640108`.

[31] Stephen Breen. What Do WebLogic, WebSphere, JBoss, Jenkins, OpenNMS, and Your Application Have in Common? This Vulnerability. `https://goo.gl/cx7X4D`, 2015. [Online; accessed 5-February-2020].

[32] bsonspec. Wala: T.j. watson libraries for analysis. `http://wala.sf.net`.

[33] Jacob Burnim, Sudeep Juvekar, and Koushik Sen. Wise: Automated test generation for worst-case complexity. In *Proceedings of the 31st International Conference on Software Engineering*. IEEE Computer Society, 2009. doi: 10.1109/ICSE.2009.5070545. URL `http://dx.doi.org/10.1109/ICSE.2009.5070545`.

[34] Richard Chang, Guofei Jiang, Franjo Ivancic, Sriram Sankaranarayanan, and Vitaly Shmatikov. Inputs of coma: Static detection of denial-of-service vulnerabilities. In *Proceedings CSF'09*, pages 186–199. IEEE, 2009.

[35] Cristina Cifuentes, Andrew Gross, and Nathan Keynes. Understanding caller-sensitive method vulnerabilities: a class of access control vulnerabilities in the java platform. In *Proceedings SOAP'15*, pages 7–12. ACM, 2015.

[36] James Clause, Wanchun Li, and Alessandro Orso. Dytan: a generic dynamic taint analysis framework. In *Proceedings of the 2007 international symposium on Software testing and analysis*, pages 196–206, 2007.

[37] Wouter Coekaerts. SerialDOS. `https://gist.github.com/coekie/a27cc406fc9f3dc7a70d`, 2015. [Online; accessed 14-January-2020].

[38] David M. Cohen, Siddhartha R. Dalal, Jesse Parelius, and Gardner C. Patton. The combinatorial design approach to automatic test generation. *IEEE Softw.*, 13(5):83–88, September 1996. ISSN 0740-7459. doi: 10.1109/52.536462. URL `http://dx.doi.org/10.1109/52.536462`.

[39] Common Vulnerabilities and Exposures. CVE-2003-1564 (Billion Laughs). `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2003-1564`, 2003. [Online; accessed 14-January-2020].

[40] Common Vulnerabilities and Exposures. CVE-2012-0161 (.NET Framework Serialization Vulnerability). `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2012-0161`, 2012. [Online; accessed 15-January-2019].

[41] Common Vulnerabilities and Exposures. CVE-2012-0160 (.NET Framework Serialization Vulnerability). `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2012-0160`, 2012. [Online; accessed 15-January-2019].

[42] Common Vulnerabilities and Exposures. CVE-2012-4406 (Deserialization Vulnerability in OpenStack Object Storage). `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2012-4406`, 2012. [Online; accessed 15-January-2019].

[43] Common Vulnerabilities and Exposures. CVE-2013-0269 (Denial of Service and Unsafe Object Creation Vulnerability in). `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-0269`, 2013. [Online; accessed 31-October-2016].

[44] Common Vulnerabilities and Exposures. CVE-2015-2937 (MediaWiki quadratic blowup vulnerability). `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-2937`, 2015. [Online; accessed 15-January-2019].

[45] Common Vulnerabilities and Exposures. CVE-2013-3171 (Delegate Serialization Vulnerability). `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-3171`, 2016. [Online; accessed 15-January-2019].

[46] Stefano Cristalli, Edoardo Vignati, Danilo Bruschi, and Andrea Lanzi. *Trusted Execution Path for Protecting Java Applications Against Deserialization of Untrusted Data: 21st International Symposium, RAID 2018, Heraklion, Crete,*

Greece, September 10-12, 2018, Proceedings, pages 445–464. 09 2018. ISBN 978-3-030-00469-9. doi: 10.1007/978-3-030-00470-5_21.

[47] Scott A Crosby and Dan S Wallach. Denial of service via algorithmic complexity attacks. In *Proceedings of 21th Usenix Security Symposium*, volume 2, 2003.

[48] Christoph Csallner and Yannis Smaragdakis. Jcrasher: An automatic robustness tester for java. *Softw. Pract. Exper.*, 34(11):1025–1050, September 2004. ISSN 0038-0644. doi: 10.1002/spe.602. URL `http://dx.doi.org/10.1002/spe.602`.

[49] Christoph Csallner and Yannis Smaragdakis. Check 'n' crash: Combining static checking and testing. In *Proceedings of the 27th International Conference on Software Engineering*, ICSE '05, pages 422–431, New York, NY, USA, 2005. ACM. ISBN 1-58113-963-2. doi: 10.1145/1062455.1062533. URL `http://doi.acm.org/10.1145/1062455.1062533`.

[50] Christoph Csallner, Yannis Smaragdakis, and Tao Xie. Dsd-crasher: A hybrid analysis tool for bug finding. *ACM Trans. Softw. Eng. Methodol.*, 17(2):8:1–8:37, May 2008. ISSN 1049-331X. doi: 10.1145/1348250.1348254. URL `http://doi.acm.org/10.1145/1348250.1348254`.

[51] CVE Details. CVE-2009-1190 (Algorithmic Complexity Vulnerability in java.util.regex.Pattern.compile). `http://www.cvedetails.com/cve/CVE-2009-1190/`, 2009. [Online; accessed 15-January-2019].

[52] CVE Details. CVE-2015-3253 (Vulnerability in Groovy). `https://www.cvedetails.com/cve/CVE-2015-3253/`, 2015. [Online; accessed 31-August-2018].

[53] CVE Details. CVE-2016-2510 (Vulnerability in Java Deserialization). `http://www.cvedetails.com/cve/CVE-2016-2510/`, 2016. [Online; accessed 15-January-2019].

[54] CWE674. CWE-674: Uncontrolled Recursion. `https://cwe.mitre.org/data/definitions/674.html`, 2019. [Online; accessed 15-January-2020].

[55] Johannes Dahse, Nikolai Krein, and Thorsten Holz. Code Reuse Attacks in PHP: Automated POP Chain Generation. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, pages 42–53, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2957-6. doi: 10.1145/2660267.2660363. URL `http://doi.acm.org/10.1145/2660267.2660363`.

[56] Johannes Dahse, Nikolai Krein, and Thorsten Holz. Code reuse attacks in php: Automated pop chain generation. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, pages 42–53, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2957-6. doi: 10.1145/2660267.2660363. URL `http://doi.acm.org/10.1145/2660267.2660363`.

[57] Jens Dietrich, Nicholas Hollingum, and Bernhard Scholz. Giga-scale exhaustive points-to analysis for java in under a minute. In *OOPSLA'15*. ACM, 2015.

[58] Jens Dietrich, Kamil Jezek, Shawn Rasheed, Amjed Tahir, and Alex Potanin. Evil pickles: Dos attacks based on object-graph engineering. In *ECOOP*, 2017.

[59] Ecma International. ECMAScript Language Specification, Standard ECMA-262 5.1 Edition / June 2011. `http://www.ecma-international.org/ecma-262/5.1/index.html`, 2011. [Online; accessed 14-January-2020].

[60] Ecma International. ECMAScript 2015 Language Specification, Standard ECMA-262 6th Edition / June 2015. `http://www.ecma-international.org/ecma-262/6.0/index.html`, 2015. [Online; accessed 31-October-2016].

[61] Anne Edmundson, Brian Holtkamp, Emanuel Rivera, Matthew Finifter, Adrian Mettler, and David Wagner. An empirical study on the effectiveness of security code review. In *Proceedings of the 5th International Conference on Engineering Secure Software and Systems*, ESSoS'13, pages 197–212, Berlin, Heidelberg, 2013. Springer-Verlag. ISBN 978-3-642-36562-1. doi: 10.1007/978-3-642-36563-8_14. URL `http://dx.doi.org/10.1007/978-3-642-36563-8_14`.

[62] G. Endignoux, O. Levillain, and J. Y. Migeon. Caradoc: A pragmatic approach to pdf parsing and validation. In *2016 IEEE Security and Privacy Workshops (SPW)*, pages 126–139, May 2016. doi: 10.1109/SPW.2016.39.

[63] Michael D Ernst. Static and dynamic analysis: Synergy and duality. In *Proceedings WODA'03*, 2003.

[64] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. The daikon system for dynamic detection of likely invariants. *Sci. Comput. Program.*, 69(1-3):35–45, December 2007. ISSN 0167-6423. doi: 10.1016/j.scico.2007.01.015. URL `http://dx.doi.org/10.1016/j.scico.2007.01.015`.

[65] Esoteric Software. Kryo: Java serialization and cloning: fast, efficient, automatic. `https://github.com/EsotericSoftware/kryo`, 2016. [Online; accessed 31-October-2016].

[66] Martin Johns Felderer, Matthias Büchler, Ruth Breu Achim D. Brucker, and Alexander Pretschner. Security testing: A survey, 2015. URL `https://doi.org/10.1016/bs.adcom.2015.11.003`.

[67] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for java. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, PLDI '02, pages 234–245, New York, NY, USA, 2002. ACM. ISBN 1-58113-463-0. doi: 10.1145/512529.512558. URL `http://doi.acm.org/10.1145/512529.512558`.

[68] Gordon Fraser and Andrea Arcuri. Evosuite: Automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, pages 416–419, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0443-6. doi: 10.1145/2025113.2025179. URL `http://doi.acm.org/10.1145/2025113.2025179`.

[69] Christopher Frohoff and Gabriel Lawrence. Marshalling Pickles. `http://frohoff.github.io/appseccali-marshalling-pickles/`, 2015. URL `http://frohoff.github.io/appseccali-marshalling-pickles/`. [Online; accessed 31-October-2016].

[70] Christopher Frohoff and Gabriel Lawrence. Marshalling Pickles. `http://frohoff.github.io/appseccali-marshalling-pickles/`, 2015. [Online; accessed 31-January-2020].

[71] Erich Gamma, John Vlissides, Ralph Johnson, and Richard Helm. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley, 1994.

[72] Vijay Ganesh, Tim Leek, and Martin Rinard. Taint-based directed whitebox fuzzing. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 474–484, Washington, DC, USA, 2009. IEEE Computer Society. ISBN 978-1-4244-3453-4. doi: 10.1109/ICSE.2009.5070546. URL `http://dx.doi.org/10.1109/ICSE.2009.5070546`.

[73] GhostScript. GhostScript, An interpreter for the PostScript language and for PDF. `https://www.ghostscript.com/`, 2019. [Online; accessed 14-January-2020].

[74] Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: Directed automated random testing. *SIGPLAN Not.*, 40(6):213–223, June 2005. ISSN 0362-1340.

doi: 10.1145/1064978.1065036. URL http://doi.acm.org/10.1145/1064978.1065036.

[75] Patrice Godefroid, Michael Y Levin, and David Molnar. Sage: whitebox fuzzing for security testing. *Communications of the ACM*, 55(3):40–44, 2012.

[76] J. A. Goguen and J. Meseguer. Security policies and security models. In *1982 IEEE Symposium on Security and Privacy*, pages 11–11, 1982. doi: 10.1109/SP.1982.10014.

[77] François Goichon, Stéphane Frénot, and Guillaume Salagnac. Exploiting java code interactions. 2011.

[78] Google. Protocol Buffers. https://developers.google.com/protocol-buffers/, 2016. [Online; accessed 30-November-2016].

[79] James Gosling, Bill Joy, Guy Steele, Gilad Brache, and Alex Buckley. The Java® Language Specification Java SE 8 Edition. https://docs.oracle.com/javase/specs/jls/se8/jls8.pdf, 2015. [Online; accessed 08-October-2020].

[80] goyaml. YAML support for the Go language. https://github.com/go-yaml/yaml, 2019. [Online; accessed 11-March-2019].

[81] Neville Grech and Yannis Smaragdakis. P/taint: Unified points-to and taint analysis. *Proc. ACM Program. Lang.*, 1(OOPSLA), October 2017. doi: 10.1145/3133926. URL https://doi.org/10.1145/3133926.

[82] Neville Grech, George Fourtounis, Adrian Francalanza, and Yannis Smaragdakis. Heaps don't lie: countering unsoundness with heap snapshots. *Proceedings of the ACM on Programming Languages*, (OOPSLA), 2017.

[83] David Grove, Greg DeFouw, Jeffrey Dean, and Craig Chambers. Call graph construction in object-oriented languages. In *Proceedings of the 12th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '97, pages 108–124, New York, NY, USA, 1997. ACM. ISBN 0-89791-908-4. doi: 10.1145/263698.264352. URL http://doi.acm.org/10.1145/263698.264352.

[84] Gu and Liu. Denial of Service Attacks. https://s2.ist.psu.edu/paper/ddos-chap-gu-june-07.pdf, 2015. [Online; accessed 5-November-2016].

[85] Ian Haken. Automated Discovery of Deserialization Gadget Chains, 2018. URL https://i.blackhat.com/us-18/Thu-August-9/us-18-Haken-Automated-Discovery-of-Deserialization-Gadget-Chains-wp.pdf. [Online; accessed 25-May-2020].

[86] Mario Heiderich, Tilman Frosch, Meiko Jensen, and Thorsten Holz. Crouching tiger - hidden payload: Security risks of scalable vectors graphics. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, CCS '11, pages 239–250, New York, NY, USA, 2011. Association for Computing Machinery. ISBN 9781450309486. doi: 10.1145/2046707.2046735. URL `https://doi.org/10.1145/2046707.2046735`.

[87] Maurice P Herlihy and Barbara Liskov. A value transmission method for abstract data types. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(4):527–551, 1982.

[88] Benjamin Holland, Ganesh Ram Santhanam, Payas Awadhutkar, and Suresh Kothari. Statically-informed dynamic analysis tools to detect algorithmic complexity vulnerabilities. In *Proceedings SCAM'16*. IEEE, 2016.

[89] Philipp Holzinger, Stefan Triller, Alexandre Bartel, and Eric Bodden. An in-depth study of more than ten years of java exploitation. In *Proceedings CCS'16*. ACM, 2016.

[90] M. Hschele and A. Zeller. Mining input grammars from dynamic taints. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 720–725, 2016.

[91] Karthick Jayaraman, David Harvison, Vijay Ganesh, and Adam Kiezun. jfuzz: A concolic whitebox fuzzer for java. In *NASA Formal Methods*, 2009.

[92] Andreas Jedlitschka, Marcus Ciolkowski, and Dietmar Pfahl. *Reporting Experiments in Software Engineering*, pages 201–228. Springer London, London, 2008. ISBN 978-1-84800-044-5. doi: 10.1007/978-1-84800-044-5_8. URL `https://doi.org/10.1007/978-1-84800-044-5_8`.

[93] jmh. Java Microbenchmarking Harness (JMH). `http://openjdk.java.net/projects/code-tools/jmh/`, 2016. [Online; accessed 5-February-2020].

[94] jsonspec. Introducing JSON. `https://json.org`, 2019. [Online; accessed 15-January-2019].

[95] jsonspec. JavaBeans API Specification. `https://download.oracle.com/otn-pub/jcp/7224-javabeans-1.01-fr-spec-oth-JSpec/beans.101.pdf`, 2020. [Online; accessed 15-January-2020].

[96] jsyaml. JavaScript YAML parser and dumper. `https://github.com/nodeca/js-yaml`, 2019. [Online; accessed 11-March-2019].

[97] Vini Kanvar and Uday P. Khedker. Heap abstractions for static analysis. *ACM Comput. Surv.*, 49(2):29:1–29:47, June 2016. ISSN 0360-0300. doi: 10.1145/2931098. URL http://doi.acm.org/10.1145/2931098.

[98] Rody Kersten, Kasper Luckow, and Corina S. Păsăreanu. Poster: Afl-based fuzzing for java with kelinci. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, pages 2511–2513, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-4946-8. doi: 10.1145/3133956.3138820. URL http://doi.acm.org/10.1145/3133956.3138820.

[99] James C. King. Symbolic execution and program testing. *Commun. ACM*, 19 (7):385–394, July 1976. ISSN 0001-0782. doi: 10.1145/360248.360252. URL http://doi.acm.org/10.1145/360248.360252.

[100] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. Evaluating fuzz testing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, pages 2123–2138, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450356930. doi: 10.1145/3243734.3243804. URL https://doi.org/10.1145/3243734.3243804.

[101] Dexter Kozen. *Language-Based Security*, pages 284–298. Springer Berlin Heidelberg, Berlin, Heidelberg, 1999. ISBN 978-3-540-48340-3. doi: 10.1007/3-540-48340-3_26. URL http://dx.doi.org/10.1007/3-540-48340-3_26.

[102] Caroline Lemieux, Rohan Padhye, Koushik Sen, and Dawn Song. Perffuzz: Automatically generating pathological inputs. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2018, pages 254–265, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450356992. doi: 10.1145/3213846.3213874. URL https://doi.org/10.1145/3213846.3213874.

[103] Johannes Lerch, Ben Hermann, Eric Bodden, and Mira Mezini. Flowtwist: efficient context-sensitive inside-out taint analysis for large codebases. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 98–108, 2014.

[104] libyaml. LibYAML - A C library for parsing and emitting YAML. https://github.com/yaml/libyaml, 2019. [Online; accessed 11-March-2019].

[105] Benjamin Livshits, Manu Sridharan, Yannis Smaragdakis, Ondřej Lhoták, J. Nelson Amaral, Bor-Yuh Evan Chang, Samuel Z. Guyer, Uday P. Khedker, Anders

Møller, and Dimitrios Vardoulakis. In Defense of Soundiness: A Manifesto. *Commun. ACM*, 58(2):44–46, January 2015. ISSN 0001-0782. doi: 10.1145/2644805. URL `http://doi.acm.org/10.1145/2644805`.

[106] Benjamin Livshits, Manu Sridharan, Yannis Smaragdakis, Ondřej Lhoták, J Nelson Amaral, Bor-Yuh Evan Chang, Samuel Z Guyer, Uday P Khedker, Anders Møller, and Dimitrios Vardoulakis. In defense of soundiness: a manifesto. *Communications of the ACM*, 58(2):44–46, 2015.

[107] V. Benjamin Livshits and Monica S. Lam. Finding security vulnerabilities in java applications with static analysis. In *Proceedings of the 14th Conference on USENIX Security Symposium - Volume 14*, SSYM'05, page 18, USA, 2005. USENIX Association.

[108] Florian D. Loch, Martin Johns, Martin Hecker, Martin Mohr, and Gregor Snelting. Hybrid taint analysis for java ee. In *Proceedings of the 35th Annual ACM Symposium on Applied Computing*, SAC '20, page 17161725, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450368667. doi: 10.1145/3341105.3373887. URL `https://doi.org/10.1145/3341105.3373887`.

[109] Fred Long. Software vulnerabilities in java. Technical Report CMU/SEI-2005-TN-044, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 2005. URL `http://resources.sei.cmu.edu/library/asset-view.cfm?AssetID=7573`.

[110] Lei Ma, Cyrille Artho, Cheng Zhang, Hiroyuki Sato, Johannes Gmeiner, and Rudolf Ramler. Grt: Program-analysis-guided random testing (t). In *Proceedings of the 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 212–223. IEEE Computer Society, 2015.

[111] Valentin J. M. Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J. Schwartz, and Maverick Woo. Fuzzing: Art, science, and engineering. *CoRR*, abs/1812.00140, 2018. URL `http://arxiv.org/abs/1812.00140`.

[112] marshalsec. Introducing JSON. `https://github.com/mbechler/marshalsec/blob/master/marshalsec.pdf`, 2020. [Online; accessed 15-January-2019].

[113] G. McGraw. Software security. *IEEE Security Privacy*, 2(2):80–83, Mar 2004. ISSN 1540-7993. doi: 10.1109/MSECP.2004.1281254.

[114] Sun Microsystems. Sun Microsystems Java Security Overview. `http://www.oracle.com/technetwork/java/js-white-paper-149932.pdf`, 2015. URL

http://www.oracle.com/technetwork/java/js-white-paper-149932.pdf. [Online; accessed 5-November-2016].

[115] Ana Milanova, Atanas Rountev, and Barbara G. Ryder. Parameterized object sensitivity for points-to analysis for java. *ACM Trans. Softw. Eng. Methodol.*, 14 (1):1–41, January 2005. ISSN 1049-331X. doi: 10.1145/1044834.1044835. URL http://doi.acm.org/10.1145/1044834.1044835.

[116] Barton P. Miller, Louis Fredriksen, and Bryan So. An empirical study of the reliability of unix utilities. *Commun. ACM*, 33(12):32–44, December 1990. ISSN 0001-0782. doi: 10.1145/96267.96279. URL http://doi.acm.org/10.1145/96267.96279.

[117] Heather Miller, Philipp Haller, Eugene Burmako, and Martin Odersky. Instant pickles: generating object-oriented pickler combinators for fast and extensible serialization. In *Proceedings OOPSLA'13*, pages 183–202. ACM, 2013.

[118] Anders Møller and Michael I Schwartzbach. Static program analysis. *Notes. Feb*, 2012.

[119] A. Muñoz and C. Schneider. The Perils of Java Deserialization. https://community.hpe.com/t5/Security-Research/The-perils-of-Java-deserialization/ba-p/6838995#.WECzUsJ96cY, 2016. [Online; accessed 1-December-2016].

[120] Alvaro Muñoz. Serial Killer: Silently Pwning Your Java Endpoints. https://www.rsaconference.com/writable/presentations/file_upload/asd-f03-serial-killer-silently-pwning-your-java-endpoints.pdf, 2016. [Online; accessed 3-December-2016].

[121] Andrew C. Myers. Jflow: Practical mostly-static information flow control. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '99, pages 228–241, New York, NY, USA, 1999. ACM. ISBN 1-58113-095-3. doi: 10.1145/292540.292561. URL http://doi.acm.org/10.1145/292540.292561.

[122] Changhai Nie and Hareton Leung. A survey of combinatorial testing. *ACM Computing Surveys (CSUR)*, 43(2):11, 2011.

[123] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999. ISBN 3540654100.

[124] Adrian Nistor, Linhai Song, Darko Marinov, and Shan Lu. Toddler: Detecting performance problems via similar memory-access patterns. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 562–571. IEEE Press, 2013.

[125] Yannic Noller, Rody Kersten, and Corina S Păsăreanu. Badger: complexity analysis with fuzzing and symbolic execution. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 322–332, 2018.

[126] Oswaldo Olivo, Isil Dillig, and Calvin Lin. Static detection of asymptotic performance bugs in collection traversals. In *Proceedings PLDI'15*, pages 369–378. ACM, 2015.

[127] openapi. OpenAPI Specification. `https://github.com/OAI/OpenAPI-Specification`, 2019. [Online; accessed 11-March-2019].

[128] OpenJDK. JEP 290: Filter Incoming Serialization Data. `http://openjdk.java.net/jeps/290`, 2016. [Online; accessed 14-January-2020].

[129] Oracle. The Java Virtual Machine Specification Java SE 8 Edition. `https://docs.oracle.com/javase/specs/jvms/se8/jvms8.pdf`, 2015. URL `https://docs.oracle.com/javase/specs/jvms/se8/jvms8.pdf`. [Online; accessed 06-July-2017].

[130] Catherine Oriat. Jartege: a tool for random generation of unit tests for java classes. *CoRR*, abs/cs/0412012, 2004. URL `http://arxiv.org/abs/cs/0412012`.

[131] Carlos Pacheco and Michael D. Ernst. Eclat: Automatic generation and classification of test inputs. In *Proceedings of the 19th European Conference on Object-Oriented Programming*, ECOOP'05, pages 504–527, Berlin, Heidelberg, 2005. Springer-Verlag. ISBN 3-540-27992-X, 978-3-540-27992-1. doi: 10.1007/11531142_22. URL `http://dx.doi.org/10.1007/11531142_22`.

[132] Carlos Pacheco and Michael D. Ernst. Randoop: Feedback-directed random testing for java. In *Companion to the 22Nd ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications Companion*, OOPSLA '07, pages 815–816, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-865-7. doi: 10.1145/1297846.1297902. URL `http://doi.acm.org/10.1145/1297846.1297902`.

[133] Carlos Pacheco and Michael D. Ernst. Randoop: Feedback-directed random testing for java. In *Companion to the 22Nd ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications Companion*, OOPSLA '07, pages 815–816, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-865-7. doi: 10.1145/1297846.1297902. URL `http://doi.acm.org/10.1145/1297846.1297902`.

[134] Rohan Padhye and Koushik Sen. Travioli: a dynamic analysis for detecting data-structure traversals. In *Proceedings of the 39th International Conference on Software Engineering*, pages 473–483. IEEE Press, 2017.

[135] Rohan Padhye, Caroline Lemieux, Koushik Sen, Mike Papadakis, and Yves Le Traon. Semantic fuzzing with zest. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2019, pages 329–340, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450362245. doi: 10.1145/3293882.3330576. URL `https://doi.org/10.1145/3293882.3330576`.

[136] Matthew Papi, Mahmood Ali, Telmo Correa Jr, Jeff Perkins, and Michael Ernst. Practical pluggable types via the checker framework.

[137] Corina S. Păsăreanu, Rody Kersten, Kasper Luckow, and Quoc-Sang Phan. Chapter six - symbolic execution and recent applications to worst-case execution, load testing, and security analysis. volume 113 of *Advances in Computers*, pages 289–314. Elsevier, 2019. doi: 10.1016/bs.adcom.2018.10.004. URL `http://www.sciencedirect.com/science/article/pii/S0065245818300640`.

[138] pdfspec. PDF Reference 6th edition. `https://www.adobe.com/content/dam/acom/en/devnet/pdf/pdf_reference_archive/pdf_reference_1-7.pdf`, 2006. [Online; accessed 14-January-2020].

[139] Or Peles and Roee Hay. One class to rule them all: 0-day deserialization vulnerabilities in android. In *Proceedings WOOT'15*. USENIX, 2015.

[140] perlsyck. YAML::Syck and JSON::Syck. `https://github.com/toddr/YAML-Syck`, 2019. [Online; accessed 11-March-2019].

[141] perlxs. Perl binding to libyaml. `https://github.com/ingydotnet/yaml-libyaml-pm`, 2019. [Online; accessed 11-March-2019].

[142] Theofilos Petsios, Jason Zhao, Angelos D. Keromytis, and Suman Jana. Slowfuzz: Automated domain-independent detection of algorithmic complexity vulnerabilities. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, pages 2155–2168, New York, NY,

USA, 2017. Association for Computing Machinery. ISBN 9781450349468. doi: 10.1145/3133956.3134073. URL https://doi.org/10.1145/3133956.3134073.

[143] phpinternals. PHP Internals. http://www.phpinternalsbook.com/, 2019. [Online; accessed 11-March-2019].

[144] phppeclyaml. PHP YAML Extension. https://pecl.php.net/package/yaml, 2019. [Online; accessed 11-March-2019].

[145] phpsymfony. CVE-2014-9130 scanner.c in LibYAML 0.1.5 and 0.1.6, as used in YAML-XS module for Perl, allows denial of service. https://www.cvedetails.com/cve/CVE-2014-9130/, 2014. [Online; accessed 11-March-2019].

[146] phpsymfony. CVE-2018-13043 Debian devscripts using unsafe YAML Loading . https://nvd.nist.gov/vuln/detail/CVE-2018-13043, 2019. [Online; accessed 11-March-2019].

[147] phpsymfony. CVE-2017-18342 PyYAML could execute arbitrary code. https://nvd.nist.gov/vuln/detail/CVE-2017-18342, 2019. [Online; accessed 11-March-2019].

[148] phpsymfony. Symfony Yaml component. https://symfony.com/doc/current/components/yaml.html, 2019. [Online; accessed 11-March-2019].

[149] M. Pistoia, S. Chandra, S. J. Fink, and E. Yahav. A survey of static analysis methods for identifying security vulnerabilities in software systems. *IBM Syst. J.*, 46(2):265–288, April 2007. ISSN 0018-8670. doi: 10.1147/sj.462.0265. URL http://dx.doi.org/10.1147/sj.462.0265.

[150] Tomas Polesovsky. Java Deserialization Denial-of-Service Payloads. http://topolik-at-work.blogspot.co.nz/2016/04/java-deserialization-dos-payloads.html, 2016. [Online; accessed 31-October-2016].

[151] M. Pradel and T. R. Gross. Leveraging test generation and specification mining for automated bug detection without false positives. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 288–298, 2012. doi: 10.1109/ICSE.2012.6227185.

[152] pyyaml. Canonical source repository for PyYAML. https://github.com/yaml/pyyaml, 2019. [Online; accessed 11-March-2019].

[153] Shawn Rasheed and Jens Dietrich. A Hybrid Analysis to Detect Java Serialisation Vulnerabilities. In *Proc. ASE'20*. IEEE, 2020.

[154] Michael Reif, Michael Eichberg, Ben Hermann, Johannes Lerch, and Mira Mezini. Call graph construction for java libraries. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 474–486. ACM, 2016.

[155] Thomas Reps, Susan Horwitz, and Mooly Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '95, page 4961, New York, NY, USA, 1995. Association for Computing Machinery. ISBN 0897916921. doi: 10.1145/199448.199462. URL `https://doi.org/10.1145/199448.199462`.

[156] Roger Riggs, Jim Waldo, Ann Wollrath, and Krishna Bharat. Pickling state in the java system. In *Proceedings COOTS'96*. USENIX, 1996.

[157] rustyaml. A pure Rust YAML implementation. `https://github.com/chyh1990/yaml-rust`, 2019. [Online; accessed 11-March-2019].

[158] B. G. Ryder. Constructing the call graph of a program. *IEEE Transactions on Software Engineering*, SE-5(3):216–226, May 1979. ISSN 0098-5589. doi: 10.1109/TSE.1979.234183.

[159] Len Sassaman, Meredith L. Patterson, Sergey Bratus, and Michael E. Locasto. Security applications of formal language theory. *IEEE Systems Journal*, 7(3): 489–500, 2013. doi: 10.1109/JSYST.2012.2222000.

[160] Prateek Saxena, Steve Hanna, Pongsin Poosankam, and Dawn Song. Flax: Systematic discovery of client-side validation vulnerabilities in rich web applications. In *NDss*, 2010.

[161] Fred B. Schneider, J. Gregory Morrisett, and Robert Harper. A language-based approach to security. In *Informatics - 10 Years Back. 10 Years Ahead.*, pages 86–101, London, UK, UK, 2001. Springer-Verlag. ISBN 3-540-41635-8. URL `http://dl.acm.org/citation.cfm?id=647348.724331`.

[162] Bernhard Scholz, Herbert Jordan, Pavle Subotiundefined, and Till Westmann. On fast large-scale program analysis in datalog. In *Proceedings of the 25th International Conference on Compiler Construction*, CC 2016, pages 196–206, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450342414. doi: 10.1145/2892208.2892226. URL `https://doi.org/10.1145/2892208.2892226`.

[163] Marc Schönefeld. *Refactoring of Security Antipatterns in Distributed Java Components*. Schriften aus der Fakultät Wirtschaftsinformatik und Angewandte Informatik der Otto-Friedrich-Universität Bamberg. University of Bamberg Press, 2010. ISBN 9783923507689.

[164] Edward J Schwartz, Thanassis Avgerinos, and David Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *2010 IEEE symposium on Security and privacy*, pages 317–331. IEEE, 2010.

[165] serdeyaml. Strongly typed YAML library for Rust. `https://github.com/dtolnay/serde-yaml`, 2019. [Online; accessed 11-March-2019].

[166] Hovav Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, CCS '07, pages 552–561, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-703-2. doi: 10.1145/1315245.1315313. URL `http://doi.acm.org/10.1145/1315245.1315313`.

[167] Yannis Smaragdakis, George Balatsouras, George Kastrinis, and Martin Bravenboer. More sound static handling of java reflection. In Xinyu Feng and Sungwoo Park, editors, *Programming Languages and Systems*, pages 485–503, Cham, 2015. Springer International Publishing. ISBN 978-3-319-26529-2.

[168] Yannis Smaragdakis, George Balatsouras, George Kastrinis, and Martin Bravenboer. More sound static handling of java reflection. In *Asian Symposium on Programming Languages and Systems*, pages 485–503. Springer, 2015.

[169] Yannis Smaragdakis, George Balatsouras, et al. Pointer analysis. *Foundations and Trends® in Programming Languages*, 2(1):1–69, 2015.

[170] snakeyaml. snakeyaml: YAML 1.1 parser and emitter for Java. `https://bitbucket.org/asomov/snakeyaml`, 2019. [Online; accessed 11-March-2019].

[171] Christopher Späth, Christian Mainka, Vladislav Mladenov, and Jörg Schwenk. SoK: XML Parser Vulnerabilities. In *Proceedings WOOT'16*. USENIX, 2016.

[172] Rastislav Šrámek. *Uncertain Optimization Using Approximation Sets. An Algorithmic Prospective*. PhD thesis, ETH Zurich, 2013.

[173] Manu Sridharan, Denis Gopan, Lexin Shan, and Rastislav Bodík. Demand-driven points-to analysis for java. In *Proc. OOPSLA'05*. ACM, 2005.

[174] Cristian-Alexandru Staicu and Michael Pradel. Freezing the web: A study of ReDoS vulnerabilities in JavaScript-based web servers. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 361–376, 2018.

[175] Arshan Dabirsiaghi Stefano Di Paola. Expression Language Injection. `https://www.mindedsecurity.com/fileshare/ExpressionLanguageInjection.pdf`, 2016. URL `https://www.mindedsecurity.com/fileshare/ExpressionLanguageInjection.pdf`. [Online; accessed 20-May-2018].

[176] Li Sui, Jens Dietrich, Michael Emery, Shawn Rasheed, and Amjed Tahir. On the soundness of call graph construction in the presence of dynamic language features - a benchmark and tool evaluation. In Sukyoung Ryu, editor, *Programming Languages and Systems*, pages 69–88, Cham, 2018. Springer International Publishing. ISBN 978-3-030-02768-1.

[177] Li Sui, Jens Dietrich, Amjed Tahir, and George Fourtounis. On the Recall of Static Call Graph Construction in Practice. In *Proceedings of the 42nd International Conference on Software Engineering*. IEEE, to appear.

[178] Bryan Sullivan. Security Briefs - XML Denial of Service Attacks and Defenses. In *MSDN Magazine Blog*, volume 24, 2009.

[179] Vijay Sundaresan, Laurie Hendren, Chrislain Razafimahefa, Raja Vallée-Rai, Patrick Lam, Etienne Gagnon, and Charles Godin. Practical virtual method call resolution for java. *SIGPLAN Not.*, 35(10):264–280, October 2000. ISSN 0362-1340. doi: 10.1145/354222.353189. URL `https://doi.org/10.1145/354222.353189`.

[180] Vijay Sundaresan, Laurie Hendren, Chrislain Razafimahefa, Raja Vallée-Rai, Patrick Lam, Etienne Gagnon, and Charles Godin. Practical virtual method call resolution for java. *SIGPLAN Not.*, 35(10):264–280, October 2000. ISSN 0362-1340. doi: 10.1145/354222.353189. URL `http://doi.acm.org/10.1145/354222.353189`.

[181] svgspec. Scalable Vector Graphics (SVG) 1.1 (2nd edition). `https://www.w3.org/TR/SVG11/REC-SVG11-20110816.pdf`, 2011. [Online; accessed 14-January-2020].

[182] swaggercve. Arbitrary code execution via Swagger YAML parser (CVE-2017-1000207 and CVE-2017-1000208). `https://perma.cc/4PGB-LZBC`, 2017. [Online; accessed 11-March-2019].

[183] N. Tillmann and J. de Halleux. Pex. `https://research.microsoft.com/Pex`, 2019. [Online; accessed 15-January-2019].

[184] Frank Tip and Jens Palsberg. Scalable propagation-based call graph construction algorithms. *SIGPLAN Not.*, 35(10):281–293, October 2000. ISSN 0362-1340. doi: 10.1145/354222.353190. URL `http://doi.acm.org/10.1145/354222.353190`.

[185] Omer Tripp, Marco Pistoia, Stephen J. Fink, Manu Sridharan, and Omri Weisman. Taj: Effective taint analysis of web applications. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 87–97, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-392-1. doi: 10.1145/1542476.1542486. URL `http://doi.acm.org/10.1145/1542476.1542486`.

[186] Omer Tripp, Marco Pistoia, Patrick Cousot, Radhia Cousot, and Salvatore Guarnieri. Andromeda: Accurate and scalable security analysis of web applications. In *Proceedings of the 16th International Conference on Fundamental Approaches to Software Engineering*, FASE'13, pages 210–225, Berlin, Heidelberg, 2013. Springer-Verlag. ISBN 978-3-642-37056-4. doi: 10.1007/978-3-642-37057-1_15. URL `http://dx.doi.org/10.1007/978-3-642-37057-1_15`.

[187] Katrina Tsipenyuk, Brian Chess, and Gary McGraw. Seven pernicious kingdoms: a taxonomy of software security errors. *IEEE Security Privacy*, 3(6):81–84, Nov 2005. ISSN 1540-7993. doi: 10.1109/MSP.2005.159.

[188] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot: A java bytecode optimization framework. In *CASCON First Decade High Impact Papers*, pages 214–224. IBM Corp., 2010.

[189] w3c. Extensible Markup Language (XML) 1.0 (Fifth Edition). `https://www.w3.org/TR/xml/`, 2008. URL `https://www.w3.org/TR/xml/`. [Online; accessed 06-July-2017].

[190] Jiayi Wei, Jia Chen, Yu Feng, Kostas Ferles, and Isil Dillig. Singularity: Pattern fuzzing for worst case complexity. In *Proceedings of Symposium on the Foundations of Software Engineering*, ESEC/FSE 2018, pages 213–223. ACM, 2018. ISBN 978-1-4503-5573-5. doi: 10.1145/3236024.3236039. URL `http://doi.acm.org/10.1145/3236024.3236039`.

[191] Mark Weiser. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering*, ICSE '81, pages 439–449, Piscataway, NJ, USA, 1981. IEEE Press. ISBN 0-89791-146-6. URL `http://dl.acm.org/citation.cfm?id=800078.802557`.

[192] Claes Wohlin. *Experimentation in software engineering. [electronic resource].* Springer, 2012. ISBN 9783642290442. URL `http://ezproxy.massey.ac.nz/login?url=http://search.ebscohost.com/login.aspx?direct=true&db=cat00245a&AN=massey.b3130054&site=eds-live&scope=site`.

[193] Valentin Wüstholz, Oswaldo Olivo, Marijn J. Heule, and Isil Dillig. Static Detection of DoS Vulnerabilities in Programs That Use Regular Expressions. In *Proceedings, Part II, of the 23rd International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 3–20, 2017. ISBN 978-3-662-54579-9. doi: 10.1007/978-3-662-54580-5_1. URL `https://doi.org/10.1007/978-3-662-54580-5_1`.

[194] XStream. Xstream, a simple library to serialize objects to xml and back again. `http://x-stream.github.io/`, 2016. [Online; accessed 31-October-2016].

[195] yamlspec. YAML Ain't Markup Language (YAML) Version 1.2 . `https://yaml.org/spec/1.2/spec.html`, 2019. [Online; accessed 14-January-2020].

[196] Mihalis Yannakakis. Graph-theoretic methods in database theory. In *Proceedings of the Ninth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, PODS '90, pages 230–242, New York, NY, USA, 1990. Association for Computing Machinery. ISBN 0897913523. doi: 10.1145/298514.298576. URL `https://doi.org/10.1145/298514.298576`.

[197] Michal Zalewski. American Fuzzy Lop (AFL). `http://lcamtuf.coredump.cx/afl/technical_details.txt`, 2017. URL `http://lcamtuf.coredump.cx/afl/technical_details.txt`. [Online; accessed 20-May-2018].

Appendix A

Statement of Contribution

MASSEY UNIVERSITY
TE KUNENGA KI PUREHUROA
UNIVERSITY OF NEW ZEALAND

GRADUATE RESEARCH SCHOOL

STATEMENT OF CONTRIBUTION
DOCTORATE WITH PUBLICATIONS/MANUSCRIPTS

We, the candidate and the candidate's Primary Supervisor, certify that all co-authors have consented to their work being included in the thesis and they have accepted the candidate's contribution as indicated below in the *Statement of Originality*.

Name of candidate:	SHAWN RASHEED
Name/title of Primary Supervisor:	CATHERINE MCCARTIN
In which chapter is the manuscript /published work:	CHAPTER 5

Please select one of the following three options:

○ The manuscript/published work is published or in press

- Please provide the full reference of the Research Output:

◉ The manuscript is currently under review for publication – please indicate:

- The name of the journal:

30th USENIX Security Symposium

- The percentage of the manuscript/published work that was contributed by the candidate: 90.00
- Describe the contribution that the candidate has made to the manuscript/published work:

Survey related work, implementation and conducting analysis, evaluation of results, reporting security issues.

○ It is intended that the manuscript will be published, but it has not yet been submitted to a journal

Candidate's Signature:	Shawn Rasheed
Date:	26-Feb-2020
Primary Supervisor's Signature:	Catherine McCartin
Date:	01/03/2021

This form should appear at the end of each thesis chapter/section/appendix submitted as a manuscript/ publication or collected as an appendix at the end of the thesis.

MASSEY UNIVERSITY
TE KUNENGA KI PUREHUROA
UNIVERSITY OF NEW ZEALAND

GRADUATE RESEARCH SCHOOL

STATEMENT OF CONTRIBUTION
DOCTORATE WITH PUBLICATIONS/MANUSCRIPTS

We, the candidate and the candidate's Primary Supervisor, certify that all co-authors have consented to their work being included in the thesis and they have accepted the candidate's contribution as indicated below in the *Statement of Originality*.

Name of candidate:	SHAWN RASHEED
Name/title of Primary Supervisor:	CATHERINE MCCARTIN
In which chapter is the manuscript /published work:	CHAPTER 6

Please select one of the following three options:

(●) The manuscript/published work is published or in press

- Please provide the full reference of the Research Output:

The 18th IEEE International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom 2019)

( ) The manuscript is currently under review for publication – please indicate:

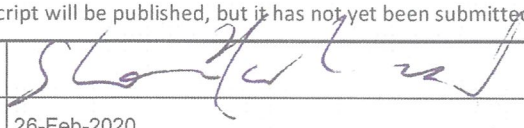
- The name of the journal:

- The percentage of the manuscript/published work that was contributed by the candidate: 90.00

- Describe the contribution that the candidate has made to the manuscript/published work:

Survey related work, gathering and preparing dataset, construction of test vectors, running experiment, evaluation of results, designing and implementing mitigation measure, reporting security issues.

( ) It is intended that the manuscript will be published, but it has not yet been submitted to a journal

Candidate's Signature:	
Date:	26-Feb-2020
Primary Supervisor's Signature:	Catherine McCartin.
Date:	01/03/2021

This form should appear at the end of each thesis chapter/section/appendix submitted as a manuscript/ publication or collected as an appendix at the end of the thesis.

MASSEY UNIVERSITY
TE KUNENGA KI PUREHUROA
UNIVERSITY OF NEW ZEALAND

GRADUATE RESEARCH SCHOOL

STATEMENT OF CONTRIBUTION
DOCTORATE WITH PUBLICATIONS/MANUSCRIPTS

We, the candidate and the candidate's Primary Supervisor, certify that all co-authors have consented to their work being included in the thesis and they have accepted the candidate's contribution as indicated below in the *Statement of Originality*.

Name of candidate:	SHAWN RASHEED
Name/title of Primary Supervisor:	CATHERINE MCCARTIN
In which chapter is the manuscript /published work:	CHAPTER 7

Please select one of the following three options:

(●) The manuscript/published work is published or in press

- Please provide the full reference of the Research Output:

The 35th IEEE/ACM International Conference on Automated Software Engineering (ASE 2020)

( ) The manuscript is currently under review for publication – please indicate:

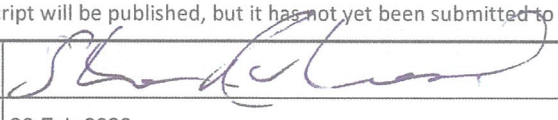
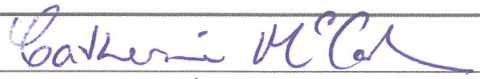
- The name of the journal:

- The percentage of the manuscript/published work that was contributed by the candidate: 90.00

- Describe the contribution that the candidate has made to the manuscript/published work:

Survey related work, specifying and implementing analysis, gathering and preparing dataset, running experiment, evaluation of results.

( ) It is intended that the manuscript will be published, but it has not yet been submitted to a journal

Candidate's Signature:	
Date:	26-Feb-2020
Primary Supervisor's Signature:	
Date:	01/03/2021

This form should appear at the end of each thesis chapter/section/appendix submitted as a manuscript/ publication or collected as an appendix at the end of the thesis.