Andreas Mili, BSc

Interactive Application Security Testing of JVM Web Applications

Master's Thesis

to achieve the university degree of

Diplom-Ingineur

Master's degree programme: Softwareengineering and Business Management

submitted to

Graz University of Technology

Advisor
Dipl.-Ing Herbert Leitold
Dipl.-Ing Dominik Ziegler

Evaluator
Univ.-Prof. Dipl-Ing. Dr.techn. Stefan Mangard

Institute of Applied Information Processing and Communications

Graz, May 2019

This document is set in Palatino, compiled with pdfLaTeX2e and Biber.

The LaTeX template from Karl Voit is based on KOMA script and can be found online: https://github.com/novoid/LaTeX-KOMA-template

Affidavit

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the sources used. The text document uploaded to TUGRAZonline is identical to the present master's thesis.

Date	Signature

Abstract

With the increasing importance of the Industrial Internet of Things (IIoT), industrial cloud platforms emerged. However, vulnerabilities in the provided services may allow an adversary to manipulate or extract sensitive data. In order to identify vulnerabilities, different security testing methods, such as Interactive Application Security Testing (IAST), have been proposed. A particular IAST approach relies on including a magic value in HTTP requests. This value is checked in so-called potentially unsafe operations, which can be adversely used when invoked with user input. However, input manipulations can limit the detection capabilities of this approach, leading to undetected vulnerabilities. As a result, they may be adversely exploited and can eventually cause harm to the industry.

In this thesis, we improve the capabilities of IAST with regard to the detection of invocations of potentially unsafe operations. Our solution tracks input manipulations by combining the aforementioned magic value approach with dynamic taint analysis at character-level. Furthermore, we automatically taint strings which are created from the magic value. We refer to this approach as interactive tainting. In order to taint primitive arrays as well, we rely on the object tagging capabilities of the JVM Tool Interface (JVMTI). With a proof-of-concept implementation for the Java Virtual Machine (JVM) platform, we verify the practicability of our concept. We show that combining taint analysis with the magic value approach of IAST offers two main advantages. First, we can detect manipulated input when it is passed to potentially unsafe operations. Second, we reduce the dependency on a comprehensive definition of user input sources by interactive tainting. In conclusion, more vulnerabilities can be detected and ultimately potential harm to the industry is prevented. Therefore, we propose other IAST tools to incorporate taint analysis.

Kurzfassung

Das Industrielle Internet der Dinge (IIoT) gewinnt zunehmend an Bedeutung und dadurch entstanden industrielle Cloud-Plattformen. Schwachstellen in den bereitgestellten Diensten können es einem Angreifer jedoch ermöglichen, sensible Daten zu manipulieren oder zu extrahieren. Um Schwachstellen zu identifizieren, wurden verschiedene Sicherheitstestmethoden, wie zum Beispiel Interactive Application Security Testing (IAST) entwickelt. Ein bestimmter IAST Ansatz beruht darauf, einen magischen Wert in HTTP Anfragen zu inkludieren. Dieser Wert wird in sogenannten potenziell unsicheren Operationen überprüft, welche bei Aufruf mit Benutzereingaben schadhaft verwendet werden können. Durch Manipulationen der Benutzeringaben kann jedoch die Erkennungsfunktion dieses Ansatzes eingeschränkt werden, wodurch nicht erkannte Schwachstellen auftreten. Infolgedessen können Operationen nachteilig ausgenutzt werden und der Industrie möglicherweise Schaden zufügen.

In dieser Arbeit verbessern wir das Potential von IAST hinsichtlich der Erkennung von Aufrufen von potenziell unsicheren Operationen. Unsere Lösung verfolgt Eingabe-Manipulationen, indem sie den oben genannten magischen Wert-Ansatz mit dynamischer Taint-Analyse auf Zeichenebene kombiniert. Außerdem markieren wir automatisch Strings, die aus dem magischen Wert erstellt werden. Wir bezeichnen diesen Ansatz als interaktives Tainten. Um auch primitive Arrays zu markieren, setzen wir die Objekt-Tagging-Funktion des JVM Tool Interface (JVMTI) ein. Mit einer Proof-of-Concept-Implementierung für die Java Virtual Machine (JVM) Plattform überprüfen wir die Umsetzbarkeit unseres Konzepts. Wir zeigen, dass die Kombination der Taint-Analyse mit dem magischen Wertansatz von IAST zwei Hauptvorteile bietet. Zum einen können wir manipulierte Eingaben erkennen, wenn sie an potenziell unsichere Operationen übergeben werden. Zum anderen reduzieren wir die Abhängigkeit von einer umfassenden

Definition von Benutzereingabequellen durch interaktives Tainting. Zusammenfassend kann festgestellt werden, dass mit unserem Ansatz weitere Schwachstellen erkannt werden und letztendlich potenzieller Schaden für die Industrie verhindert werden kann. Daher empfehlen wir andere IAST Tools, Taint-Analyse zu integrieren.

Contents

List of Figures

Glossary

AOP	Aspect-Oriented Programming.
AOT	Ahead-of-Time.
API	Application Programming Interface.
AST	Abstract Syntax Tree.
CFG	Control Flow Graph.
CLI	Command Line Interface.
CVE	Common Vulnerabilities and Exposures.
CWE	Common Weakness Enumeration.
DAST	Dynamic Application Security Testing.
DIFA	Dynamic Information Flow Analysis.
DTA	Dynamic Taint Analysis.
ear	Enterprise Archive.
HTTP	Hypertext Transfer Protocol.
IAST	Interactive Application Security Testing.
IFA	information-flow analysis.
IIoT	Industrielle Internet der Dinge.
IIoT	Industrial Internet of Things.
IoT	Internet of Things.
IV	Input Vector.
jar	Java Archive.

JDK	Java Development Kit.
JIT	Just-in-Time.
JVM	Java Virtual Machine.
JVMTI	JVM Tool Interface.
MITRE	Massachusetts Institute of Technology Research Establishment.
NIST	National Institute of Standards and Technology.
OS	Operating System.
OWASP	Open Web Application Security Project.
QA	Quality Assurance.
SAST	Static Application Security Testing.
SDLC	Software Development Lifecycle.
SMT	Satisfiability Module Theory.
SQL	Structured Query Language.
SQLI	SQL Injection.
TTLB	Time To Last Byte.
war	Web Archive.
WSDL	Web Service Description Language.
XSS	Cross-Site Scripting.
YAML	YAML Ain't Markup Language.

1 Introduction

In recent years, there has been a shift towards cloud computing in industrial environments. The primary reason for this shift is the application of the Internet of Things (IoT) in industry, also known as Industrial Internet of Things (IIoT) or *Industry 4.0* [Gil16]. The IIoT is, similar to the IoT, a network of interconnected *things*. These *things* are devices which may include sensors. They perform tasks including data collection, data upload and device control. To fulfil these tasks, they are typically connected to a cloud service. As a result, cloud platforms providing services for the IIoT emerged. These services may process sensitive data uploaded by IIoT devices, such as domain knowledge and intellectual property. Individuals like cybercriminals or competitors may be interested in such data. To preserve the confidentiality and integrity of this data, it is crucial that applications are free from security bugs. Otherwise, adversaries can exploit them to perform unauthorised actions. These actions primarily undermine or break confidentiality and integrity. Broken confidentiality allows attackers to steal intellectual property. Broken integrity, on the other hand, allows attackers to perform unwanted manipulations of data. For example, such manipulations may be used to sabotage manufacturing processes. In summary, web applications are a valuable target for attackers. Keeping them free of security bugs is a very challenging task, since bugs have been part of software programs since their origins. Thus, establishing sophisticated inspection techniques is of utmost importance. Otherwise, vulnerabilities may stay undetected, constituting a potential harm to the industry.

1.1 Motivation

Since insecure web applications are a potential threat to the industry, they need to be inspected for vulnerabilities. This task is the purpose of security testing. By improving inspection mechanisms, more vulnerabilities can be fixed, and thus the security can be improved. Inspection can be performed in different ways. Depending on the presence of the application source code, we can distinguish between white-box and black-box security testing methods. White-box methods test the inner structure of an application by analysing the source code. Black-box methods, on the other hand, have no access to the source code or any other information about the application except the public Application Programming Interface (API).

Typically, there exist two approaches of application security testing: Static Application Security Testing (SAST) [Garc] and Dynamic Application Security Testing (DAST) [Gara]. SAST, is performed white-box. It scans for vulnerabilities by inspecting the source code of the application. This inspection is done without executing the application itself. DAST of web applications is, according to Gartner [Gara], typically performed black-box by analysing the responses of a running application.

Both SAST and DAST have their weaknesses. The fact that SAST inspects source code leads to higher code coverage. Nevertheless, high code coverage does not necessarily imply a high detection rate. A study [GP15] on the detection capabilities of static analysis tools concludes that relying solely on static analysis leaves a notably amount of vulnerabilities undetected. Furthermore, SAST is also more prone to false positives since the findings are based on heuristics. A significant amount of false positives can become counterproductive very quickly and decrease the usefulness of a tool. This situation can eventually lead to the ignorance or cutoff of tests and thus provide a false sense of security.

In DAST for web applications, vulnerabilities are typically detected based on errors included in a response. If a response contains no indications of an error, it is typically assumed that no faults occurred inside the application. However, this assumption may lead to undetected vulnerabilities. The absence of error messages does not necessarily imply the absence of vulnerabilities. Hence, the detection capabilities of DAST strongly depend

on the responses of the application. This dependency becomes an issue when vulnerabilities are blind. Blind in this context refers to the nontriviality or inability to infer vulnerabilities by interpreting responses. Blind vulnerabilities can be introduced both intended or unintended. For instance, let us consider an unintended silent catch statement in the program code. This circumstance may turn an easily detectable vulnerability into a blind one. On the other hand, intentionally wrapping specific error messages into generic ones may also introduce blind vulnerabilities. For example, a Structured Query Language (SQL) exception is thrown inside an application, but a generic Hypertext Transfer Protocol (HTTP) error 404 is returned. In doing so, it is not possible to tell whether a resource was not found or a real exception occurred. Therefore, it is particularly challenging to detect such vulnerabilities. If there is no way of producing different responses, observing timing delays is the only option. However, this technique depends heavily on a good side channel. If such a side channel is not available, a vulnerability becomes fully blind.

As a result, a third approach, known as Interactive Application Security Testing (IAST) [Garb] is employed. The goal of IAST is to reduce the weaknesses of DAST. To do so, black-box and white-box approaches are combined. A vulnerability scanner represents the black-box approach. To extend its detection capabilities, the code of an application is instrumented. This instrumentation process represents the white-box approach. With this combination, a black-box scanner can gather valuable information about a running application, such as data flow and the actual configuration. Data flow is of particular interest when it concerns user input. This input can flow into operations which can be adversely used and thus are potentially unsafe. In this case, vulnerabilities arise.

The main advantage of IAST is its capability to detect fully blind vulnerabilities. A typical approach for detection performed by existing IAST solutions is to include a magic value in the response. Instrumented code checks if this value is passed to a potentially unsafe operation. If this is the case, a vulnerability is discovered. This vulnerability is then reported back to the black-box scanner. This approach works regardless of whether the vulnerability is blind or not. However, it has one major drawback. If the application manipulates the user input, it destroys the magic value. If this manipulation occurs before the magic value reaches a potentially unsafe operation,

vulnerabilities are not detected.

We conclude that the detection of potentially unsafe operations by security testing is necessary to ensure secure web applications. Powerful testing methods are required to perform this task. IAST is such a method, but input manipulations may severely limit detection capabilities. Taint analysis is an approach which can reduce this limitation. In taint analysis, the information that input originates from a user is stored as metadata. If the input is modified, the metadata is transformed accordingly. In this way, the chance that input manipulations destroy the magic taint value is reduced. Primitive taint analysis approaches store taint information for entire strings. As a consequence, a string can either be fully tainted or not tainted at all. Thus, a long untainted string, for instance, becomes tainted just by a tainted string of one character. A substring of the previously untainted part is now tainted. Such cases are likely to introduce false positives. By tracking taint information at character-level, false positives can be reduced. Currently, there is no IAST solution that integrates with web applications and has the following properties:

- Transformed inputs are taken into account
- Taint information is tracked at character-level
- Primitive arrays are taken into account

1.2 Proposed Solution

In this thesis, we develop an IAST tool which eliminates limitations of existing solutions. In particular, we present an approach which integrates well with web applications as well as web application containers.

The main contributions of our solution are:

- **Reduce Input Manipulation Effects**
  We apply dynamic taint analysis to reduce the effects of input manipulations. Such manipulations limit the detection capabilities of the IAST approach which uses magic values to mark user input. To perform taint analysis, we instrument Java Development Kit (JDK) classes which hold strings. During instrumentation, we add a field which

stores the taint information. Furthermore, we add code which propagates taint information to methods which perform manipulations on the string. Although primitive arrays are considered as objects on the Java Virtual Machine (JVM) platform, they cannot be instrumented. Therefore, we rely on the object tagging capabilities of the JVM Tool Interface (JVMTI).

- **Minimize False Positives**
  We apply two countermeasures to reduce false positives. The first one is to track taint information at a character level. For every individual character in a string, we hold information whether it is tainted or not. If partly tainted user input reaches a potentially unsafe operation, it can be evaluated in greater detail. The second countermeasure we apply is to keep track of applied sanitisation functions. For instance, it may be required for certain functionality that users invoke potentially unsafe operations. To prevent potential damage from such invocations, user input is sanitised. By keeping track which sanitisations functions have been invoked, we reduce false positives.
- **JVM Language Independence**
  Our solution is applicable for all available JVM languages. We instrument the intermediate code, also referred to as Java Bytecode, which the JVM executes. We evaluated several instrumentation frameworks and chose ByteBuddy for our purpose. Using this framework, we are able to keep our approach as generic as possible.
- **No Interference with Existing Code & Performance**
  Our approach reduces the risk of side effects by not altering existing bytecode instructions. Transformation of existing bytecode instructions increases the likelihood to introduce a different behaviour compared to the unmodified application. Furthermore, it is more likely to produce invalid bytecode. Therefore, we only add new bytecode instructions. More specifically, new instructions are added at the beginning or end of a method's instructions. In this way, we do not interfere with existing instructions. This approach also reduces the time needed for instrumentation.
- **Configurability & Extendability**
  Our solution is configurable and extensible to the fast-changing nature of software libraries. If a library introduces a new potentially unsafe operation or a new string operation is added, taint propagation is

affected. Therefore, we support customisation in two forms. First, we define configuration files for sources of user input, sanitisation functions and potential operations. These configurations support a very abstract and generic way of matching those functions. Second, we abstract the taint propagation in a way that makes it simple to add further policies.

2 Security Testing of Web Applications

In this chapter, we explain the basic concepts of security testing for web applications. Web applications process potentially sensitive data, making them a valuable target for attackers. In general, security testing aims at finding vulnerabilities which threat the confidentiality, authenticity or integrity of this data. Furthermore, also securing the availability of an application is part of security testing. Breaking any of these properties can have severe consequences. For instance, lack of confidentiality can result in the exposure of sensitive data. In case of missing authenticity, an attacker can impersonate a legitimate user and perform actions on their behalf. Broken integrity can result in data being forged or corrupted. Degraded availability becomes devastating when critical infrastructure becomes unavailable.

One of the main reasons why web applications can be insecure is that they handle user input. This input may be malicious and thus should not be trusted. Consequently, applications must be tested for robustness against such inputs. However, tests at a functional level often do not cover such inputs. They send acceptable inputs to check if requirements are met. However, this form of testing is insufficient for security testing. Unexpected and unacceptable inputs have to be taken into account as well.

There exist several technologies for security testing. In this chapter, we describe the most common techniques for web applications. Section 2.1 describes which types of vulnerabilities are common for web applications. Section 2.2 explains the security testing methodologies Static Application Security Testing (SAST), Dynamic Application Security Testing (DAST) and Interactive Application Security Testing (IAST).

2.1 Vulnerabilities

Vulnerabilities are a particular kind of software bug. They may allow an attacker to perform unauthorised actions. Vulnerabilities typically arise due to insecure configuration or improper handling of user input. Different kinds of software programs are susceptible to different vulnerabilities. In our work, we focus on web applications. Typical vulnerabilities of such applications are summarised by the Open Web Application Security Project (OWASP)[1]. Moreover, other categorisations exist, such as Common Weakness Enumeration (CWE)[2]. We shortly introduce those categorisations in the following section. Furthermore, we explain potentially unsafe operations, since our goal is to detect their invocations.

2.1.1 Categorisation

As web applications became more feature rich, the number of different vulnerability increased. Therefore, efforts have been made to categorise vulnerabilities. OWASP issue the most prominent vulnerability categorisation for web applications. With their *Top 10*[3] project, they list a broad consensus of the ten most widespread type of vulnerabilities. A much more fine-grained categorisation than OWASP *Top 10* is the CWE. It was established by the Massachusetts Institute of Technology Research Establishment (MITRE) to introduce a common language for weakness identification and categorisation. In comparison to the OWASP *Top Ten*, CWE does not only focus on web applications. It consists of over a thousand entries which are of different abstraction levels. The most dangerous weaknesses are summarised in their *Top 25*[4].

Vulnerabilities can also be categorised based on occurrences in real software. For this purpose, the National Institute of Standards and Technology (NIST) introduced the Common Vulnerabilities and Exposures (CVE) list [MIT19].

[1] https://www.owasp.org
[2] https://cwe.mitre.org/
[3] https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project
[4] https://cwe.mitre.org/top25

Figure 2.1: A typical form of SQL injection. User input is concatenated with a SQL query.

It contains vulnerabilities of specific versions of software or libraries. Most vulnerability scanners utilise this library as they scan specific entries of it.

2.1.2 Potentially Unsafe Operations

A potentially unsafe operation is any operation which can be adversely used when invoked with user input. Therefore, those operations are not unsafe per se, assuming a non-adversarial developer. Operations, like sending commands to an interpreter or reading from a file, are commonly performed in the application. In this thesis, we mainly focus on one aspect of potentially unsafe operations, namely injections. Hence, we discuss injections in the following paragraph.

Injections Whenever user input is interpreted as a command, injection attacks may occur. They are the most highly ranked type of vulnerabilities. The OWASP *Top 10* rank injections in the first place. The top two items of the CWE *Top 25* are injections as well. More specifically, Structured Query Language (SQL) injection and Operating System (OS) command injection.

Although many types of injection attacks exist, they typically follow a similar principle. Input sent by a user is used as input to an interpreter used in the application. If user input is not filtered, an adversary can send arbitrary commands to the interpreter. However, before input reaches the interpreter, it may be inserted in a predefined string of instructions, such as an SQL query or an OS Command Line Interface (CLI) command. The

inserted location is referred to as the context. As a result, it may be necessary to break out of the semantical context, to successfully inject commands. In this case, the user input has to be crafted accordingly. After context breakout, any desired commands can be injected. An example injection attack for SQL is depicted in Figure 2.1. The *'OR 1=1* – payload sent by the attacker is mixed with the predefined SQL query in the application. To break out of the context, a single quote is used. This quote ends the password search string and allows the attacker to insert further commands. In this case, the tautology *OR 1=1* is used to bypass a password check.

Consequences of injections are manifold. The potential damage depends on the commands an interpreter supports. For instance, CLI interpreter commands for user management can be used to compromise a server. For SQL injections, Halfond et al. [HVO08] have classified different types of attacks, including data extraction, authentification bypass and privilege escalation.

2.2 Methodologies

Three approaches exit for application security testing: Static Application Security Testing (SAST) [Garc], Dynamic Application Security Testing (DAST) [Gara] and Interactive Application Security Testing (IAST) [Garb]. SAST detects vulnerabilities without executing the application itself. In contrast, DAST is performed by interacting with the running application. IAST enhances the capabilities of DAST by approaches of SAST.

2.2.1 Static Application Security Testing

SAST is an approach which detects vulnerabilities without executing the application itself. SAST approaches typically analyse the source or compiled code to detect vulnerabilities. For Java Virtual Machine (JVM) applications, most approaches focus on bytecode analysis. Popular approaches include points-to analysis [LL05], taint analysis [Cao+17] and symbolic execution

[CF10]. The first two approaches can be used to detect invocations of potentially unsafe operations. Symbolic execution can be utilised to solve constraints on strings to bypass input filters.

The advantage of SAST is the ease of setup and code coverage. For the usage of SAST tools, no application deployment or setup is necessary. Therefore, those tools can be integrated early in the Software Development Lifecycle (SDLC). Compared to DAST and IAST, the code coverage of static testing is the most comprehensive. When all dependencies of an application are supplied, exhaustive path coverage can be reached.

However, analysing code statically also introduces drawbacks. Because the application is not tested in its running state, assumptions about its behaviour have to be made. For instance, security-relevant configurations are often only known at runtime. Similarly, access control policies or interactions with external systems are typically not known at compile-time. A further consequence of SAST is false positives. Those can arise from by over-approximation of a program's behaviour, as pointed out by Trinh et al. [TCJ14]. Finally, a study [GP15] indicates that relying on SAST alone leads to undiscovered vulnerabilities. Therefore, high coverage does not necessarily imply a high detection rate.

2.2.2 Dynamic Application Security Testing

Dynamic Application Security Testing (DAST) represents the counterpart of SAST. It analyses an application during runtime. In web applications, vulnerabilities are detected by analysing the responses.

This analysis is typically performed in a black-box setting. In this setting, tools have no access to the source code or any other information about the application except the public Application Programming Interface (API). Furthermore, also white-box and grey-box settings exist. In a white-box setting, full access to source code is available. A grey-box setting typically gives access to an application's configurations or specifications. In the following, we discuss two typical approaches used in dynamic testing.

Fuzz Testing. One popular approach to generate faulty data is fuzz testing. The majority of approaches rely on injecting certain faults into the application. The responses to this faulty data are observed for anomalies. In this way, vulnerabilities are discovered. Fuzzying is performed black-box, grey-box or white-box. When performing fuzzing in a black-box setting, achieving a high code coverage is challenging. If conditionals are not satisfied by the data, paths may never be taken. In this case, false negatives arise.

Dynamic Taint Analysis. Taint analysis tries to solve the so-called tainted object propagation problem. It comprises sources, sinks and optionally sanitisers. Sources are functions from which user inputs originates. Sinks represent potentially unsafe operations. Sanisiters transform user input in a form that is not harmful to a particular sink. To solve the tainted object propagation problem, one must detect if user input from a source can reach a sink. Several steps have to be performed for detection. First, user input is marked as tainted, since it may be malicious. Next, taint information is propagated. Taint propagation ensures that taint information is properly passed when new variables are derived from tainted ones. Finally, instrumentation code in sinks checks if tainted user input is present in security-relevant parameters.

Vulnerability Scanner. Several methodologies, including the above mentioned, are combined in tools. These tools are often referred to as vulnerability scanners. Open-source, as well as commercial variants, are available. Well-known open-source scanners include ZAP[5], Arachni[6] and Nikto[7]. Commercial scanners include Burp Suite[8], Acunetix[9], AppScan[10]. According to Hope and Walther [HW08], two components are typical for such scanners, namely so-called spider and test-cases for well-known vulnerabilities.

[5] https://github.com/zaproxy/zaproxy
[6] https://www.arachni-scanner.com/
[7] https://cirt.net/Nikto2
[8] https://portswigger.net
[9] https://www.acunetix.com/
[10] https://www.ibm.com/us-en/marketplace/appscan-standard

Spiders can be seen as crawlers whose goal is to identify the attack surface. This process includes parsing responses for links and following them. Forms and parameters are identified as well. After this process, test-cases for well-known vulnerabilities are executed. This phase is referred to as scanning.

Advantages and Disadvantages. The advantage of DAST is typically, according to several authors [Dou+12; GGS14] a lower false positive rate compared to SAST. Since DAST tests an application in its running state, behaviour does not need to be approximated.

The major drawback of DAST is a typically lower detection rate in comparison to SAST. There are two reasons for this circumstance. First, paths in which vulnerabilities are present may not be executed. There have been efforts to explore more paths with techniques like fuzzing [LLW14; Duc+14] and a combination of fuzzing with symbolic execution [GLM08]. Nevertheless, full coverage is impractical because programs can have an unbounded number of paths. The second reason for a lower detection rate is blind vulnerabilities. These types of vulnerabilities depend on a side channel and are practically not detectable if no exploitable side channel exists.

2.2.3 Interactive Application Security Testing

Interactive Application Security Testing (IAST) is a form of testing which incorporates both SAST and DAST. However, instead of applying both techniques separately, it creates a synergy between them. The importance of this synergy has been pointed out first by Ernst [Ern03]. They suggested blending the strengths of static and dynamic approaches. Information collected by SAST can be utilised by DAST and vice versa.

Two typical forms of IAST approaches exist. First, information collected by static code analysis is utilised. Such information includes potential Input vectors (IVs) and constraints which restrict payloads. Second, information collected from a running application is utilised. We further refer to such approaches as agent-based approaches. These agents contain algorithms which

gather information from the running application, such as invoked SQL functions. In the following, we describe one approach for each category.

String Constraint Solving. Im et al. [IYJ17] proposed an IAST approach which incorporates static code analysis to address filters in applications. Filters restrict payloads and therefore also attack strings of dynamic analysers. Bypassing filters can lead to a higher detection rate. Therefore, they try to solve the constraints of those filters. In this way, attack strings bypassing those filters can be generated. They developed an interaction platform which incorporates static and dynamic analysers. The static analyser is used to gather string constraints. These constraints are solved using a string constraint solver.

Magic Value Tainting. A typical agent-based approach to enhance DAST capabilities is what we refer to as *magic value tainting*. The goal of this approach is to detect the invocations of potentially unsafe operations with user input. The detection is achieved by the interaction between instrumented code and the DAST scanner. First, the DAST scanner sends requests which include magic value. Instrumentation code then checks for this value in invocations of potentially unsafe operations. If the value is found, it is certain that user input can reach a potentially unsafe operation. Subsequently, detected vulnerabilities are reported back to the DAST scanner.

Vendors of security testing tools started to incorporate IAST agents. Portswigger[11] use this approach in their *Infiltrator* component. Acunetix[12] developed an agent as well, called *Acusensor*. Both vendors employ a magic value tainting approach.

Advantages and Disadvantages. Since IAST uses information extracted from code, higher detection rates compared to DAST can be achieved. A particular weakness of DAST is that it does not recognise invocations of potentially unsafe operations via response analysis. The agent-based

[11] https://portswigger.net
[12] https://www.acunetix.com/

approach of IAST can detect such invocations. Furthermore, it can report the exact line of vulnerable code. As a result, developers can fix vulnerabilities quicker and thus deploy security updates in a more timely manner.

The advantages which are gained through code analysis and instrumentation introduce a dependency on source or bytecode. Without code access, IAST capabilities cannot be leveraged. Furthermore, as with DAST, coverage still remains an issue with IAST. Although more paths can be explored with IV detection and string constraint solving, full coverage cannot be guaranteed. This is an inherent limitation of dynamic testing. Vulnerabilities can only be detected if the code path in which they reside is actually executed.

3 The Java Virtual Machine

In this chapter, we explain the fundamentals of the JVM platform. This information is vital to understand the instrumentation of programs developed for the JVM. The JVM is a virtual machine, designed for hardware- and OS independent programs. It abstracts the underlying hardware and OS from the programmer. Because of this abstraction, programmers do not have to take platform specific behaviour into account. The functionality of the JVM is standardised in the JVM specification [LB15]. This standard ensures that every programmer can rely on the same behaviour across different JVM implementations. The most prominent implementation is called HotSpot [Oraa]. It was developed by Sun Microsystems and is now developed and maintained by Oracle. Currently, HotSpot is available for Linux, MacOS, Windows and Solaris SPARC.

Section 3.1 describes the process of compilation of source code until the execution of native code on a high-level. This description illustrates the different locations where instrumentation code can be added. Section 3.2 describes the Java class file format in closer detail. Section 3.3 covers the execution of class files. Section 3.4 explains how programs developed for the JVM can be instrumented.

3.1 From Source Code to Native Code

In this section, we explain the process from compilation of source code to the generation of native code. This process is depicted in Figure 3.1. Programs targeted for the JVM are commonly written in a high-level language. A compiler translates this source code to an intermediate representation, so-called class files. These class files represent classes or interfaces in a

binary form. Section 3.1.1 explains the compilation process in greater detail. After compilation, the generated class files are usually packaged, which is described in Section 3.1.2. Finally, the JVM loads the class files and translates the bytecode to native instructions. Thus, programs can be written in any language supporting compilation to class files. A well-known example of such a language is Java. It makes use of the *javac* compiler to translate *.java* files to class files. Other well-known JVM languages are Kotlin, Groovy and Scala.

3.1.1 Compilation

The starting point of the compilation process is source code of a program written in a JVM language. Subsequently, a compiler is used to transform the source code into a binary representation. This stream of bytes is stored in class files. The structure of a class file is explained in greater detail in Section 3.2. As an alternative to the compilation to class files, Java 9 introduced Ahead-of-Time (AOT) compilation. The goal of AOT is to speed up the startup of JVM applications. By compiling source code immediately to native code, time for interpretation and Just-in-Time (JIT) compilation is spared. For that purpose, the AOT compiler of the graal project[1] is used. To date, AOT compilation still has several limitations. It is in an experimental state and only supports Linux x64 systems. Furthermore, classes which use dynamically generated code like lambda expressions cannot be compiled ahead-of-time.

3.1.2 Packaging

Class files are typically packaged into zip files. Well-known formats are Java Archive (jar), Web Archive (war) and Enterprise Archive (ear). Jar files follow a specific format, described in the jar file specification [Orab]. If a jar file contains all dependent classes which are referenced in the class files, it is called a *fat* jar. War files are used for web applications adhering to the Java servlet specification [CB]. The archive contains an additional WEB-INF

[1] https://www.graalvm.org/

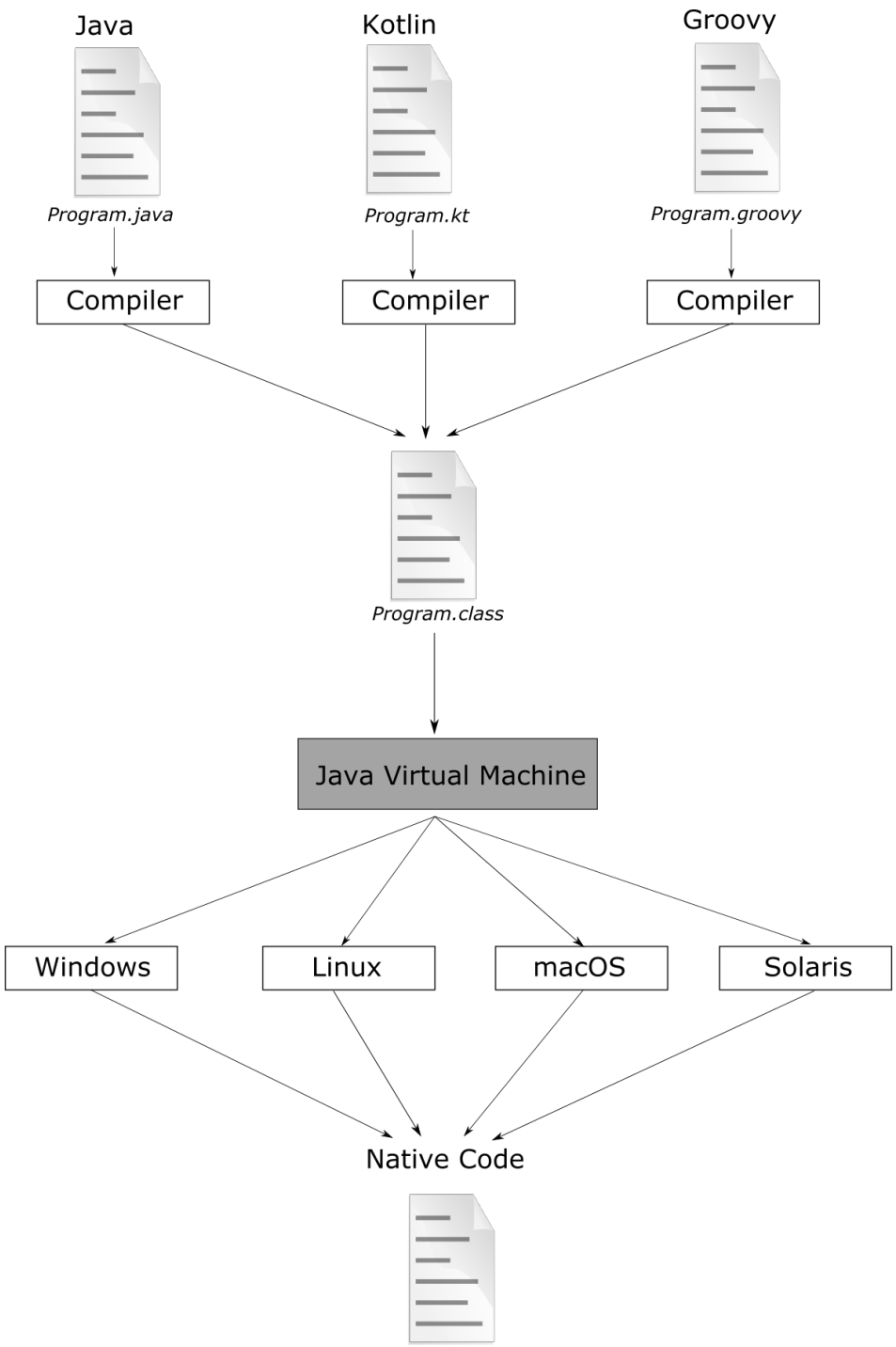

Figure 3.1: Transformation of source code to native code.

directory. This directory comprises compiled java servlet classes, libraries, and a deployment descriptor file named *web.xml*. The ear file format is used for Java enterprise applications [DS]. An Ear archive may also contain jar and war files. A common part of all archive formats is the MANIFEST.MF file in the META-INF directory. Apart from meta information about the packaged files, it contains the relative path to the main class.

Let us consider the above-explained compilation process by taking the Java language as an example. In this case, the starting point is source code written according to the Java Language Specification [Gos+18]. This source code is stored in *.java* files. Next, the *javac* compiler is used to compile the classes in the *.java* files to class files. The final step is packaging the class files into an archive. Various tools can perform this task. The Java Development Kit (JDK) for instance ships with the command line tool *jar*. Further options include build tools like Ant[2] or Gradle[3].

3.2 The Class File Format

Class files constitute a cornerstone of the JVMs platform-independence. They can be seen as an intermediate representation of source code. This representation is platform-independent and translated to native code for a specific platform by the JVM. A class file describes the structure of exactly one class or interface. If a source file contains several classes or interfaces, the compiler generates separate class files for each.

The class file format is specified in the JVM specification [LB15, Section 4] in form of a C-style struct. This format is illustrated in Listing 3.1. Every line represents the size and type of a class file section. The size is either specified in chunks of bytes or as a structure. Chunk sizes include one byte (`u1`), two bytes (`u2`) and four bytes (`u4`). Structures are composed of the aforementioned chunks, or contain structures itself. The items `cp_info`, `field_info`, `method_info` and `attribute_info` are such structures. It is important to mention that the array notation does not imply access at fixed

[2] https://ant.apache.org/
[3] https://gradle.org/

byte-offsets like in C-style arrays. The notation rather indicates a list of structures, which are of variable size. The reason for that is that structures may itself contain lists of structures.

In the following paragraphs, we describe the essential sections of the class file format in greater detail. Basic sections are described by comments in the listing itself.

Listing 3.1: The structure of a class file, according to the JVM specification [LB15]

```
ClassFile {
  u4             magic;         // magic number, always 0xCAFEBABE
  u2             minor_version; // minor class file version
  u2             major_version; // major class file version
  u2             constant_pool_count;
  cp_info        constant_pool[constant_pool_count-1];
  u2             access_flags;  // access permissions, i.e. PUBLIC
  u2             this_class;    // name of the class
  u2             super_class;   // direct super class, if existing
  u2             interfaces_count;
  u2             interfaces[interfaces_count]; // implemented
                                               // interfaces
  u2             fields_count;
  field_info     fields[fields_count];
  u2             methods_count;
  method_info    methods[methods_count];
  u2             attributes_count;
  attribute_info attributes[attributes_count];
}
```

constant_pool. This item is the cornerstone for dynamic linking. It consists of a list of entries, such as class, field or method references. Entries may have different sizes. The constant pool also contains concrete values. Such values are for instance strings denoting a class, method or field name as well as numeric constants. Furthermore, entries may also reference entries within the constant pool itself.

Entries in the constant pool are referenced by so-called *symbolic references* in other parts of the class file. Depending on the particular JVM implemen-

tation, symbolic references are either resolved when a class is loaded or at run-time. During resolution, the symbolic references are replaced with actual memory addresses. Hence, class files do not depend on the run-time memory layout of a JVM implementation.

fields. This item contains all fields which are defined in the class or interface described by the class file. Fields from superclasses or superinterfaces are not inherited. The `field_info` structure is used to describe fields. First of all, the structure includes an access modifier. The name and type of the field are described via references to the constant pool. Finally, the `field_info` structure contains attributes. The most prominent type of attributes for fields are annotations.

methods. All methods including the constructor of the class being described, are defined in this item. A `method_info` structure is used to describe a method. The first entry in the structure defines the access modifier. Further, the name, parameters and return type are described via a reference to the constant pool. Like the `field_info` structure, the structure also includes attributes. As with fields attributes, one type of them are annotations. An important type of attribute for methods is the `code_attribute`. This structure contains the bytecode instructions for non-abstract and non-native methods. Moreover, it defines the maximum number of local variables and the maximum depth of the operand stack.

attributes. Attributes at class file level are defined in this item. The referenced inner classes, annotations and the source file name are examples for such attributes.

3.3 JVM Internals

The Java Virtual Machine (JVM) executes programs which have been compiled to class files. To execute class files, the JVM has to perform several steps. First, classes have to be loaded, linked and initialised. We refer to

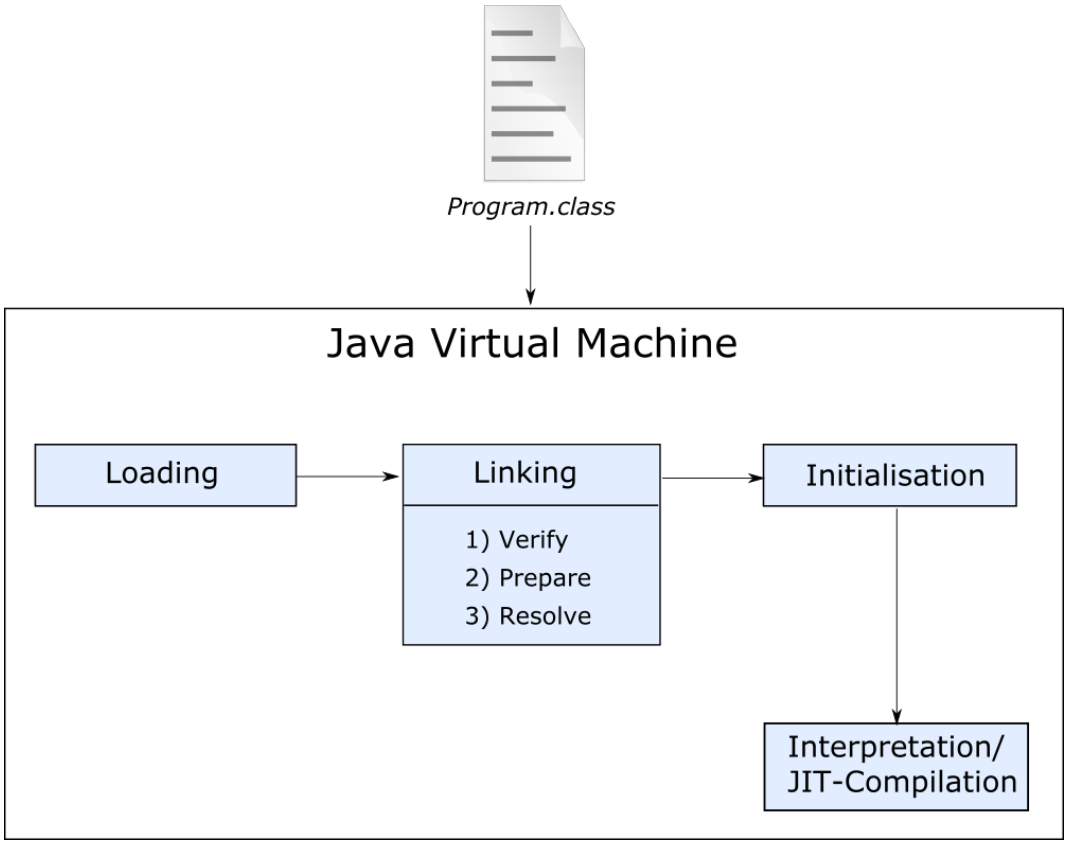

Figure 3.2: How the JVM executes a class file.

this procedure as the class loading process. Then, the interpreter and JIT compiler produce native code the JVM can execute.

Section 3.3.1 explains the class loading process with the steps loading, linking and initialisation. In Section 3.3.2, we cover details about the verification of class files. Finally, interpretation and JIT compilation is described in Section 3.3.3

3.3.1 Class Loading Process

To execute code, classes or interfaces stored in class files are first transformed into a runnable state. To achieve that, the JVM performs the stages loading, linking and initialisation. Since loading is just one part of the complete process, we refer to all stages as the class loading process.

Loading. In the loading stage, the binary representation of classes or interfaces is loaded into the JVM. A class loader is responsible for this procedure. The JVM specification [LB15] specifies two kinds of class loaders: User-defined class loaders and bootstrap class loader. A user-defined class loader must be a subclass of the abstract class *ClassLoader*. They are used to customise the behaviour of the loading process. For instance, classes can be loaded from different locations than the local filesystem. Furthermore, it is also possible to generate classes on the fly and supplying these classes to the class loader. In the end, the loader has to return a stream of bytes in the Java class file format. Since user-defined class loaders are regular classes themselves, they have to be loaded by a class loader as well. Hence, there exists a particular class loader which does not depend on the JVM for its execution. Such a particular kind of class loader is the bootstrap class loader. It is not written in any JVM language but in native code. Thus, every JVM implementation for a particular platform has its own bootstrap class loader.

Linking. After class loading, the linking stage is performed. During linking, classes are verified, prepared and resolved. Verification is an important task to ensure memory and type safety. Therefore it is explained separately in Section 3.3.2. Preparation involves the allocation of memory for all static fields and its assignment of default values. Concrete values are assigned in the initialising stage. During resolution, symbolic references in the run-time constant pool are replaced with original memory references. This step is optional and can be deferred to run-time.

Initialisation. Initialisation is the final stage, in which the initialisation method of a class or interface is executed. This method comprises all code from static initialisers. Furthermore, it includes all assignments of static variables to their concrete values.

3.3.2 Class File Verification

The JVM gives strong guarantees concerning memory and type safety. To fulfil these guarantees, class files have to be validated. The central part of the validation process involves checking the static and structural constraints of bytecode instructions. Those constraints are defined in the JVM specification [LB15]. Static constraints concern the well-formedness of the bytecode. For example, the target of a jump instruction in the bytecode of a method must be an instruction within the same method. Structural constraints are related to the relationship between bytecode instructions. For instance, it is validated whether a local variable is accessed before initialisation. In addition to the verification of methods bytecode, the class file itself is validated. When instrumenting bytecode, special care has to be taken to prevent the violation of static and structural constraints.

3.3.3 Interpretation and Just-in-Time Compilation

Intepretation, Just-in-Time (JIT) compilation, or a combination of both transform JVM bytecode instructions to native instructions. It is up to the concrete JVM implementation which techniques are used. Interpretation is performed by translating one bytecode instruction after another to native instructions. A dedicated interpreter is responsible for executing those native instructions. JIT compilation, on the other hand, translates chunks of bytecode instructions into native code. Processing larger chunks of instructions provide more possibilities for optimisations.

To provide an example, we consider the most widely used JVM implementation HotSpot [Oraa]. It is based on the observation that the most frequently executed parts of a program are just a small subset of it. Therefore, optimisation is focused on those most frequently executed parts, also called hot spots.

HotSpot features an interpreter and two distinct compilers. The interpreter matches a template of native code instructions to every bytecode instruction, as described in [HM18]. First, bytecode is executed using the interpreter, to collect statistics about frequently executed code. These statistics are used to trigger the JIT compilers. Subsequently, the generated native instructions are used instead of the interpreter. HotSpot includes a client and a server compiler. The client compiler is constructed for fast compilation with few optimisations. The server compiler performs more aggressive optimisations but takes more time. To combine the benefits of both compilers, tiered compilation was added to HotSpot. In this approach, both the client and server compiler are used for a single application. In doing so, a good compromise between fast startup and runtime performance can be achieved.

3.4 Instrumentation

The goal of instrumentation is to enrich programs with additional code, to inspect its state and behaviour. Especially in Quality Assurance (QA), it can be beneficial to know additional information about a running application's behaviour. Such information is for instance which lines of code are executed, or which functions are affected by a user-defined input. With this knowledge, the quality of testing can be improved. However, the code for collecting and providing this information is usually not included in the application by default. It is added manually by instrumenting source or bytecode.

Instrumentation may be categorised in two ways. One form of distinction is the point in time when instrumentation code is added. Static instrumentation is performed before an application is started. Dynamic instrumentation is done at application startup or runtime. Another way for categorisation is the layer of abstraction at which additional code is added. These layers are either source code or bytecode. Source code instrumentation is covered in Section 3.4.1. Instrumentation at bytecode level is explained in Section 3.4.2. State of the art bytecode instrumentation frameworks are reviewed in Section 3.4.3.

3.4.1 Source Code Instrumentation

The purpose of source code instrumentation is to enrich source code with additional behaviour. It typically works by modification of the Abstract Syntax Tree (AST). The AST is generated by a parser and represents the source code in a structured way. This representation allows it to perform efficient modifications.

Instrumenting code on source level is less complex than on bytecode level. The reason for that is, that source code is easier to understand and less verbose. However, these advantages come with less portability. Since different languages rely on the JVM, instrumentation code cannot be written generically for all JVM languages. Another limitation of source code instrumentation is that it can only be performed statically. Since the JVM requires compiled classes for execution, runtime instrumentation is not possible.

3.4.2 Bytecode Instrumentation

The goal of bytecode instrumentation is to modify the behaviour of an application by changing its bytecode. On the JVM, this approach is performed by the modification of class files. An essential part of a class file for instrumentation is the included instructions. These are stored within methods. The instruction set is defined in the JVM specification [LB15, section 6].

Bytecode instrumentation offers two main advantages. First of all, it enables instrumentation independent of the language used for the source code. Since the code is added to class files and not source files, it is irrelevant which high-level language is used. The second advantage is the possibility to perform instrumentation at runtime. This approach is useful in case not all dependencies are available for post-compile instrumentation. One such case is a thin jar, which does not include any dependencies. Instead, they are supplied separately and added to the classpath.

3.4.3 Bytecode Instrumentation Frameworks

Several bytecode instrumentation frameworks exist for the JVM platform. They mainly differ in two aspects. The first aspect is the amount of control over the instrumentation process. The amount of control ranges from the addition of code at predefined points to the arbitrary transformation of bytecode instructions. In general, frameworks with low-level APIs provide a higher amount of control compared to those with high-level APIs. The second aspect categorises frameworks according to the required knowledge of the class file format. Some instrumentation frameworks only require knowledge of a JVM language. Others require a certain degree of knowledge about class files and JVM instructions. Those are usually the frameworks which provide the highest amount of control over the instrumentation process.

Bytecode instrumentation resembles a core part of this thesis. Therefore, we have carefully evaluated the latest bytecode instrumentation frameworks. These frameworks include ASM[4], Javassist[5], Cglib[6], BCEL[7], AspectJ[8] and ByteBuddy[9].

AspectJ. AspectJ is an Aspect-Oriented Programming (AOP) framework for the JVM. In AOP, the instrumentation code is defined in so-called aspects. The process of merging the application's code with aspects is called weaving. AspectJ supports weaving before compilation, after compilation and during runtime. Using AspectJ is simple because knowledge of the class file format is not required. Aspects can be written in the application's native language. However, this advantage also introduces certain limitations. Instrumentation code can only be added at specific predefined locations, so-called joinpoints. Examples of such joinpoints are method calls or field references. To date, joinpoints for conditionals or local variable access do not exist. Furthermore, it is not possible to transform existing bytecode instructions.

[4] https://asm.ow2.io/
[5] http://www.javassist.org/
[6] https://github.com/cglib/cglib
[7] https://commons.apache.org/proper/commons-bcel/
[8] https://www.eclipse.org/aspectj/
[9] http://bytebuddy.net/

Javassist. This framework [Chioo] was initially developed to provide a low-level instrumentation API. To date, it also supports writing instrumentation code in a source language similar to Java. A custom compiler is responsible to compile the code. The apparent advantage of javassist is that it provides both a low-level and high-level API. However, the fact that the instrumentation code for the high-level API is not written in plain Java results in certain limitations. For instance, their official manual[10] states that annotations and generics are not supported.

ASM. The ASM framework [BLCo2] was built for high-performance. Programming languages like Groovy and Kotlin use ASM for compiling high-level programming constructs to bytecode. Furthermore, AspectJ and Byte-Buddy are building on top of it. The current implementation provides two low-level APIs, as described in the official documentation[11]. First, the *core* API is event-based and designed to be fast and memory efficient. During the parsing of a class file, events are generated for each element of a class. Such elements are for instance methods, fields or instructions. Then, handler for the generated events modify the classes' bytecode. The second API is called *tree* API and is built on top of the core API. It is object-based and stores the whole content of a class file in a single object.

ASM provides a high amount of control over the instrumentation process. Compared to other low-level APIs, ASM is designed to be as fast and small as possible. However, profound knowledge of the class file format is required for its usage.

ByteBuddy. ByteBuddy is a framework which allows generating and transforming classes at runtime. For that purpose, it uses the visitor API of the ASM framework. ByteBuddy provides an intuitive high-level API. Though its high-level nature, it is designed to provide a high degree of control over the instrumentation process. Fields and methods can be added, removed and to some extent also transformed. For instance, it is possible to

[10] http://www.javassist.org/tutorial/tutorial2.html#limit
[11] https://asm.ow2.io/asm4-guide.pdf

delegate method calls. Furthermore, transforming JVM instructions is also possible using the ASM core API.

3.5 Agents

Java and native agents support capabilities which are not available to standard application code. In particular, these capabilities allow programmers to modify and get information about a running JVM. For instance, profiler use agents to gather information about performance metrics like memory usage or execution time of executed methods. Agents may also be used to influence the behaviour of an application actively. Both Java and native agents provide access to classes' bytecode for instrumentation. By having this access, existing bytecode can be changed, or new instructions added.

3.5.1 Java Agents

Java agents are written purely in a JVM language. They are packaged into `jar` files like standard JVM applications. However, agents do not run in an own JVM instance. Instead, they are attached to the JVM instance of a running program. Java agents were introduced by the `java.lang.instrument` API in JDK 5. Via this API one can access the bytecode of classes' for instrumentation purposes. Thus, the primary purpose of a Java agent is to instrument bytecode. In doing so, it is possible to modify or get information about an application's behaviour.

Classes can be instrumented during or after JVM startup. For these purposes, the `java.lang.instrument` API defines the `premain` and `agentmain` functions. These have to match the signatures given in Listing 3.2.

Listing 3.2: Signatures of the `premain` and `agentmain` functions.

```
1 public static void premain(String agentArgs,
    Instrumentation inst);
2 public static void agentmain(String agentArgs,
    Instrumentation inst);
```

Instrumentation at VM startup is performed by implementing the `premain` method. The JVM executes this function before the `public static void main(String[] args)` function of the application to which the agent is attached. Therefore, the agent has to be specified during the startup of the application. In this way, it is possible to instrument classes of an application before they are loaded into the JVM. Performing instrumentation during startup has a significant advantage. It allows modifying the class file structure of a class to be loaded. Thus, additional methods and fields can be added. However, this does not apply for classes which are already loaded before `premain` is invoked. The class file structure of these so called bootstrap classes can yet not be modified, as recorded in an enhancement proposal[12]. Examples are the `java.lang.String` or `java.lang.Thread` classes. As a consequence, Java agents cannot be used out-of-the-box to add fields or methods to bootstrap classes. Nevertheless, there exist efforts the circumvent this restriction, for instance, DCVM[13] or JRebel[14]. DCVM is a modified version of the HotSpot JVM. JRebel uses a Java agent combined with abstract bytecode and classloaders. Although it is not allowed to alter the class file structure, method implementation still can be modified. This modification must be done under the premise, that the method's signature remains unchanged.

By implementing the `agentmain` method, instrumentation can be performed after VM startup. As opposed to agent attachment at startup, the `agentmain` method is invoked after the `public static void main(String[] args)` method.

As already stated, agents are packed in `jar` files like standard applications. Additionally, a manifest file need needs to be supplied. It is necessary in order for the JVM to treat the archive as an agent. The following entries are used to configure an agent:

- `Premain-Class`
  Specifies the class containing the `premain` method
- `Agent-Class`
  Specifies the class containing the `agentmain` method

[12] http://openjdk.java.net/jeps/159
[13] https://github.com/dcevm/dcevm
[14] https://jrebel.com/software/jrebel/

- `Boot-Class-Path`
  List of directories or libraries added to the bootstrap classpath
- `Can-Redefine-Classes`
  Boolean specifying the ability to redefine classes. Redefinition replaces a class by supplying a new definition in the form of a byte array.
- `Can-Retransform-Classes`
  Boolean specifying the ability to retransform classes. Retransformation is perfomed by transforming an existing definition of class. This task is handled by registering instances of so called `ClassFileTransformer`.
- `Can-Set-Native-Method-Prefix`
  Boolean specifying the ability to set native method prefixes. This mechanism allows it to instrument native methods by wrapping them in a non-native method.

3.5.2 Native Agents

Native agents operate on a different level compared to Java agents. They use low-level capabilities provided by the JVM Tool Interface (JVMTI). These capabilities can be used to inspect and control the state of a JVM. The JVMTI is defined in a C header file named `jvmti.h`. When implementing a native agent, this header file must be included. Therefore, native agents are typically written in the C/C++ language. Then, they are compiled to a shared library. Another option is to statically link the agent with the JVM. Agents are attached during JVM startup or runtime. The JVMTI defines a wide range of capabilities. These can be divided into two applications areas, namely inspection or modification of a running JVMs state.

Inspection of the JVM is mainly performed by registering and receiving events. Such events include changes of fields or methods. For instance, it is possible to register for events in case a classes' field is accessed or modified. Furthermore, one can also register for notifications when a classes' method is entered or exited. Additional features include statistics about CPU usage or heap tagging. The latter is intended to track arbitrary objects by attaching labels to them.

The JVMTI also supports actively influencing a running JVM. Debuggers

are a typical application area of native agents. Breakpoints points can be set and cleared. Threads may be suspended, resumed or stopped. Moreover, static fields of classes can be accessed and set. The JVMTI also supports a way of changing classes' bytecode. The `ClassFileLoadHook` event is used for that purpose. This event is triggered prior to the JVM loading a particular class. It provides complete access to all classes for one-time instrumentation, including bootstrap classes.

Listing 3.3: Example agent with callback at VM start.

```
1 JNIEXPORT jint JNICALL Agent_OnLoad(JavaVM *jvm,
    char *options, void *reserved) {
2   // Register capabilites
3   jvmtiCapabilities capa;
4   (void) memset(&cap, 0, sizeof(jvmtiCapabilities));
5   cap.can_tag_objects = 1;
6   error = jvmti->AddCapabilities(&cap);
7
8   // Register callbacks
9   jvmtiEventCallbacks callbacks;
10  (void) memset(&callbacks, 0, sizeof(callbacks));
11  callbacks.VMStart = &cbVMStart;
12  error = jvmti->SetEventCallbacks(&callbacks,
      (jint) sizeof(callbacks));
13 }
```

To develop a native agent, the `Agent_OnLoad` function has to be implemented. This function is invoked early in the boot phase of the JVM. On invocation, no classes or objects are loaded yet and no bytecode instructions have been executed. Listing 3.3 shows an extract of such an implementation. The desired capabilities an agent needs, have to be requested explicitly. In this example, Line 5 shows how to request the heap tagging capability. In doing so, the JVM is able to optimize its performance based on the requested capabilities. Furthermore, callbacks must be specified in order to receive events. In this example, the `JVMStart` callback is registered in Line 11.

4 Concept

This chapter elaborates how we achieve the goal of detecting invocations of potentially unsafe operations with IAST. Our concept incorporates several approaches, including code instrumentation, taint analysis and agents. In the following, we explain the design decision we made as well as the rationale behind them.

Section 4.1 gives an overview of our concept. Section 4.2 describes the components of our architecture. Section 4.3 exemplifies how the components work together to detect a potentially unsafe operation. Section 4.5 gives details about the approach we use to instrument web applications. Section 4.4 explains how we apply dynamic taint analysis.

4.1 Approach

In our concept, we focus on detecting invocations of operations, which cause potential harm when user input is passed. Throughout this document, we further refer to such operations as sinks. We detect unsafe invocations by tagging user input and checking tags when a sink is invoked. A typical IAST approach for this purpose is to include a magic value in requests of a DAST scanner. In this case, the value of the input itself gives information about its taint status. Potential vulnerabilities are then detected by checking the input parameters of potentially unsafe operations for this value. However, input manipulation can destroy this value and thus reduce detection capabilities. We counter this issue by combining this approach with taint analysis. Hereby, we instrument string-holding classes. During instrumentation, we add a field which stores taint information in these classes. When string-manipulating functions of those classes are invoked, we mimic their manipulations on the

taint information. For instance, the substring of a tainted string is tainted as well. However, primitive arrays are not covered by this approach. Therefore, we rely on the object tagging capabilities of the JVMTI with a native agent. In doing so, we can attach taint information to primitive arrays.

To perform tainting, we need to know when to attach taint information and when to check it. Therefore, we support the definition of sources and sinks. Assessing whether or not a sink is harmful is not in the scope of our work. In the field of security testing, there exist well-known function signatures of sinks. For instance, the popular static analysis tool FindSecBugs[1] includes a very comprehensive signature list. In our work, we give the user a very generic way of matching those signatures in a configuration file. Concerning the sources of user input, we are not forced to rely entirely on function signatures. In traditional taint analysis, a variable is tainted if it originates from a function with a specific signature. If such a signature is missing, user input is not tainted. We counter this problem by interacting with a black-box scanner. When a character sequence containing a magic value from the scanner is added to a string, it is automatically tainted. Hence, we can attach taint information in case of a missing source definition. Finally, we support the definition of sanitisation functions. Those are functions which transform strings to remove potential harmful character sequences.

We apply our taint concept by instrumenting the application on bytecode level. Instrumentation is performed offline, during application startup and runtime. The string-holding classes are instrumented offline. This procedure is needed since the JVM does not allow changing the class file structure after a class has been loaded. Sources, sinks and sanitisation functions are instrumented at startup or runtime. To have access to the classes for instrumentation, we use Java agents. All instrumentation code is added without transforming existing bytecode instructions. In this manner, we can minimise interferences with existing instructions.

We minimise false positives by applying two measures. First, we store taint information for every individual character in a string. In doing so, a string can contain tainted and untainted data. If taint information were stored for whole strings, an untainted string would be tainted as soon as a small tainted sequence is added. When reaching a sink, the previously untainted

[1] https://find-sec-bugs.github.io/

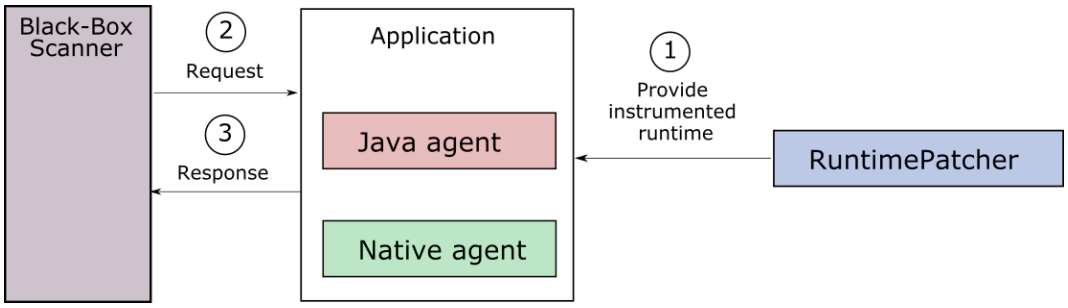

Figure 4.1: The architecture of our solution.

part cannot be distinguished anymore. With character-level tainting, it can be analysed if the tainted parts suffice to produce a vulnerability. The second measure we apply against false positives is keeping track of sanitisation functions. These functions transform a tainted string into a benign one. Thus, reporting a vulnerability for a sanitised string would produce a false positive. Therefore, we record for every string which of the defined sanitisation functions have been invoked. In sinks, it can be checked if a string was sanitised by an appropriate sanitisation function.

4.2 Architecture

In this section, we describe the overall architecture of our concept. It consists of four major components, namely a *Black-box Scanner*, a *Java agent*, a *Native Agent* and a *RuntimePatcher*. Figure 4.1 illustrates these components. In the following, we describe each of them in greater detail.

Black-box Scanner. The black-box scanner represents a party which sends requests to the application. Such a party is needed to perform IAST. In general, it can be any tool capable of sending those requests. We rely on a black-box vulnerability scanner. The scanner interacts with the application by including a magic value in its requests. This value is used in two forms: First, our agent checks for this value when a character sequence is added to a string. If the value is included in the value to be added, we taint it.

As a result, we can taint user input without predefined sources. However, the application may transform the magic value before a new string is created from it. In this case, we may lose taint information. Therefore, we still support defining sources. The second form of using the magic value is, to check for it again in sinks. Our concept is able to perform this procedure. However, we rely on taint analysis to examine the taint information associated with a string.

Java Agent. The Java agent component inserts instrumentation code for sources, sinks and sanitizers. The point in time in which a class is instrumented depends on when it is loaded. Classes which are already loaded when the agent is attached are instrumented immediately. All other classes are instrumented on first load by a classloader. This point in time can be during application startup or runtime. To perform instrumentation, those functions need to be matched. Therefore, we define matchers in configuration files. Our Java agent reads those files at startup and instruments functions accordingly.

Native Agent. The native agent is used for attaching taint information to primitive arrays. In the JDK, the `java.lang.String` class allows the programmer to get its content in form of an array of primitive characters or bytes. These arrays inherit the taint information of the string to reduce false negatives. For this purpose, we use the heap tagging capabilities of the JVMTI. With the `SetTag` and `GetTag` function, we attach taint information to primitive arrays. Therefore, we do not lose taint information if programmers extract primitive arrays.

Runtime Patcher. The runtime patcher modifies string classes. Typical classes in the JDK which store strings are: `String`, `StringBuffer` and `StringBuilder`. These classes are already loaded when our Java agent starts up. As the JVM forbids changing the class file structure of loaded classes, we perform the modification offline. We instrument the string classes by adding an attribute, storing taint information. Since we chose the attribute to be of type object, users can store taint information in any desired format.

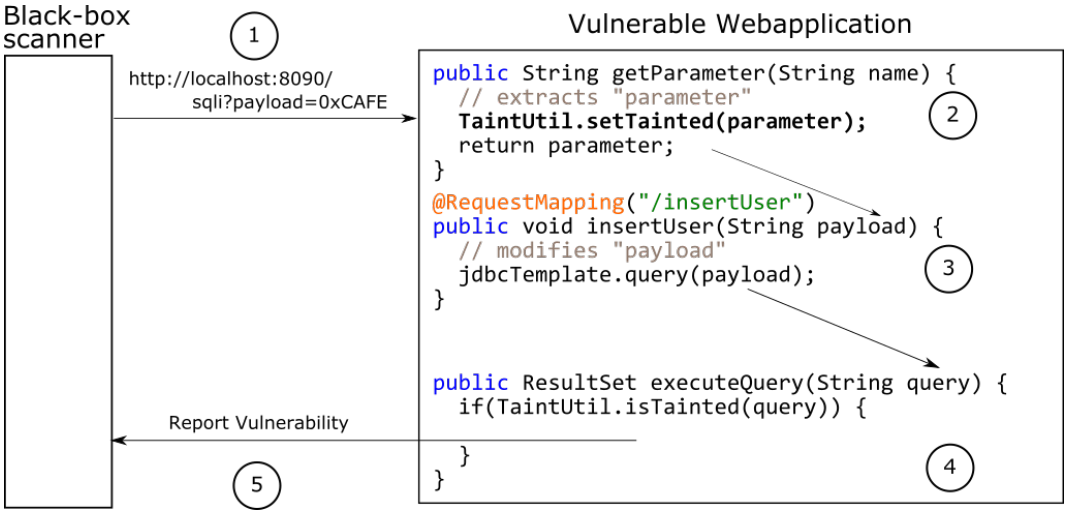

Figure 4.2: Steps required to detect a SQL injection.

Furthermore, we instrument all methods of string classes which potentially modify taint information. More specifically, we mimic the behaviour of these functions on the taint information. For instance, we append taint information if two strings are appended.

4.3 Detection Workflow Example

In this section, we illustrate how our components described in the previous section interact to detect a SQL injection vulnerability. Figure 4.2 shows the overall process of this workflow.

First, the black-box vulnerability scanner sends a request including the magic value *OxCAFE*. On the JVM, these requests are stored in classes implementing the `HttpServletRequest` interface. These classes give access to all parts of a request, like its parameters or cookies. In our example, this request will contain a parameter with the value *OxCAFE*.

In the second step, the parameter is extracted using the `getParamter(String name)` function. Therefore, this function acts as a source. Next, we can distinguish between two cases:

1. Source of user input is defined
2. Source of user input is not defined

In the first case, no matcher for the `getParameter` function is defined in the sources configuration. As a consequence, the function will not be instrumented. In our example, this means that the `TaintUtil.setTainted()` function call, depicted in bold, will be omitted. Now, the magic value *OxCAFE* plays an important role. When a request is received, byte arrays or buffers contain this magic value. These bytes are read by an open websocket of the application. Since the socket API is low-level, reading is performed bytewise. In the following, strings are created from bytes. In our example, these are strings assigned with a value including *OxCAFE*. We hence taint strings which are created from these bytes. The tainting is performed by instrumentation code added to all functions accepting character or byte arrays. This code is added by our *RuntimePatcher* component. In the second case, the sources configuration file does include a matcher for the `getParameter` function. Based on this matcher, we match the corresponding function on bytecode level for instrumentation. Both matching and instrumentation are performed by our Java agent component. During the instrumentation process, we add a method call at the end of the function. This call, in our example `TaintUtil.setTainted()`, sets the taint information attribute of the string function parameter `name`. This attribute was added to the string class in advance by the *RuntimePatcher* component, as described in the previous section. After this process, the string returned by the `getParameter` function is fully tainted.

In the third step, the tainted parts of the requests are further processed by the application. During these operations, the string may be modified by several functions. Furthermore, substrings can be taken, or the string could also be concatenated to other strings. In such cases, the taint information is propagated accordingly.

Next, tainted strings may flow into a sink, as denoted by the fourth step. In our example, such a sink is the `executeQuery()` function of the `JdbcTemplate` class. We check if the first string parameter of the function contains tainted parts. If this is the case, a potential vulnerability is discovered.

In the final step, the black-box scanner retrieves the vulnerability information. To automatically extract this information, the black-box scanner needs to be extended. In our implementation, we rely on file output to log the discovered vulnerabilities.

4.4 Taint Analysis

In this section, we describe how we use the principles of taint analysis in our system. We apply a traditional taint analysis approach relying on sources, sinks and sanitizers. Throughout this document, we refer to this approach as *non-interactive*. In addition, we enhance the non-interactive approach by reducing the dependency on a comprehensive definition of sources. We refer to this enhanced approach as *interactive tainting*. Subsequently, we will explain this approach in further detail. Furthermore, we elaborate where we store taint information, how we propagate it and how we configure sources, sinks and sanitizers.

4.4.1 Interactive Tainting

The non-interactive tainting approach relies heavily on the definition of sources. Based on this definition, the return values of these functions are tainted. In the interactive approach, we rely on a black-box scanner for tainting. As described in Section 4.3, the request of the scanner includes a magic value. After this value is stored in primitive arrays, it will typically flow into a string class. Since the value represents user input, we subsequently taint strings which are created from it.

Nevertheless, we do not solely rely on an interactive approach. We combine the interactive tainting approach with the non-interactive one. In doing so, the non-interactive and interactive approach mutually reduce their disadvantages. In particular, the possibility of missing tainting is diminished. Tainting in the interactive approach may fail if the bytes including the magic value are transformed. When a new string is created from these bytes, the magic value cannot be recognised. In this instance, a source defined in the

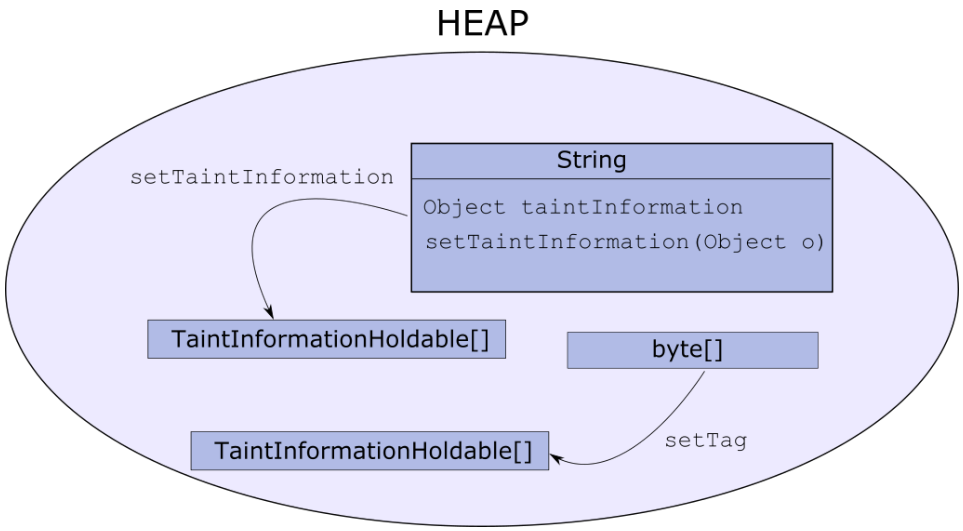

Figure 4.3: Heap with tainted objects.

non-interactive approach can still taint such a string. On the contrary, a missing source definition can lead to missed tainting. This downside is countered by the interactive approach. The worst case scenario happens when bytes are transformed, and source definition is missing. Since there already exist well-known source definitions for JVM application, this risk is, however, rather small.

4.4.2 Taint Storage Location

We store taint information in an additional field in string classes. This decision is based on two fundamental observations: First, an entirely non-invasive approach, like storing strings in a global key-value store which holds taint information, is not practical. In this case, it is vital that the strings are uniquely identifiable. The JVM specification does not specify a function for this purpose. Admittedly, one could use the `System.identityHashCode()` function included in the JDK. However, this function is not guaranteed to be free of collisions for garbage collection reasons. Our second observation is that storing taint information in the string itself requires to alter the

behaviour of all manipulating operations. More specifically, all accesses to the internal buffer containing the string need to be modified.

Storing taint information in an additional field has several advantages. For instance, we can store taint information without the need to modify existing bytecode instruction. Primitive arrays constitute an exception since their classes cannot be modified in any way. Therefore, we rely on the JVMTI to attach taint information.

String classes. We chose to store taint information in an additional field in string classes. This field is of type *Object* and thus allows the user to store taint information in any desired format. A more generic approach would be to add this field to the object class. Since primitive arrays are also objects on the JVM, they could be tainted as well. However, this approach seems to break internal assumptions of the JVM, resulting in the inability to boot. It is assumed that the reason for that is that the JVM internally accesses object fields at fixed offsets. String classes appear not to be affected by this limitation. We successfully validated our implementation against JDK version 8.

Primitive arrays. For primitive arrays, we cannot store taint in a class field. Neither the `java.lang.instrument` interface nor the JVMTI provide means to instrument those arrays. Although arrays are considered objects, the JVM dynamically creates array classes on startup. Because of this dynamic creation, it is not possible to instrument those classes. However, the JVMTI provides means to assign tags to objects on the heap. With the JVMTIs `setTag` and `getTag` functions, we can associate taint information to array objects.

Comparison. Storage-wise, the object tagging approach is similar to the approach for string classes. In both cases, the taint information is stored as an object on the heap, as illustrated in Figure 4.3. However, there is a difference in how taint information is set and where the reference to taint information is stored. For string classes, taint information is set by invoking a setter operation. In our example, this operation is called `setTaintInformation`

and was added during instrumentation. The reference to the taint information is also stored differently. For string classes, the reference is stored in string class itself. For primitive arrays, we create this link artificially with the `setTag` function.

4.4.3 Sources, Sinks and Sanitizers

We define sources, sinks and sanitizers for non-interactive taint analysis. Although sources are not strictly necessary in our interactive tainting approach, we still define them. In doing so, we can minimise the chance of missed tainting. We introduce separate configuration files for sources, sinks and sanitizers. Since configurations may be edited manually, we define a human-readable format. Therefore we rely on the human-readable language YAML Ain't Markup Language (YAML) [BEN19].

Configuration files for sources, sinks and sanitizers are comprised of so-called matchers. The purpose of a matcher is to match one or several functions. The matching is performed based on specific properties. Those properties are either on class level or function level. On a class level, we support matching based on interface, superclasses and annotations. On a function level, we support matching based on names, descriptors and annotations. The values for matchers are regular expressions. Hence, names can be matched on a generic basis.

Using a matcher, one can match specific functions or several functions at once. In some cases, it is desirable to match several functions at once based on a common property. One such property is a common super-interface. For instance, SQL statements are abstracted in the `java.sql.Statement` interface. By matching all implementations of this interface, one could generically instrument SQL related sinks, independent of the database provider used.

Functions can be matched on various levels of abstraction. Figure 4.4 exemplifies this abstraction levels on the `Files.write()` function of the JDK runtime library. Such a function is called either directly, or indirectly by a third-party library function. After calling intermediate functions, runtime library functions eventually end up calling native functions. In general,

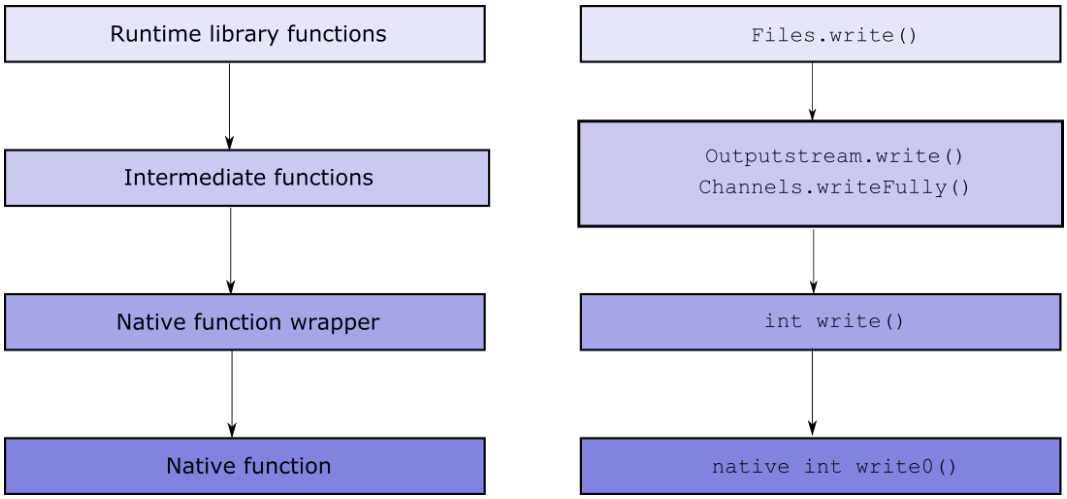

Figure 4.4: Hierarchy of a typical function call stack.

those native functions are not called directly. Instead, wrapper functions are called to provide means for instrumenting native functions.

An important question is at which abstraction level sources or sinks are defined. Matching at native method level would resemble the most generic approach. All functions using a particular native method would be included. However, for identification and mitigation purposes, knowing the caller is vital. Every level of abstraction provides more context to the function call stack. Just having the information that a particular native function acted as a source or sink is not helpful. For example, let us consider a potential vulnerability involving the `write0()` native function of the example above. To remove a vulnerability, one would either perform input sanitisation or remove the function call leading to the `write0()` call. Either way, this adaption would be performed on the same abstraction level as the initiating call. One such initiating call is the `Files.write()` runtime library function. Consequently, function definitions for sources and sinks are typically defined for runtime library functions. In doing so, programmers exactly know where code adaptions are necessary.

4.4.4 Taint Propagation

After strings are tainted, manipulating operations may be performed on them. Those operations are not automatically reflected on the taint information. Therefore, we propagate taint information manually. For propagation, we distinguish between strings, primitives and arrays of primitives. In the following, we describe how and when we propagate taint information in these cases.

Strings. String-related library classes contain several functions for manipulations. We add instrumentation code to these functions to propagate taint information. How this propagation is performed depends on the semantics of the function. For instance, if the function performs concatenation, taint information must be concatenated as well. A function that reverses a string also needs to reverse the taint information. In sum, we mimic the function logic on our taint information.

Propagation is rather simple if the string class is immutable. In the JDK, this is the case for the `java.lang.String` class. Since the length and content cannot be modified, the taint information cannot get out of sync with the string itself. However, if the string class is mutable, such as `java.lang.StringBuffer` and `java.lang.StringBuilder`, taint information may get lost.

As a result, all methods modifying the length or content of a string have to be considered for mutable string classes. In particular, problems may occur when the length of taint information differs from the string length. This difference leads to incorrect array index accesses of the taint information. As a result, the application may crash. Therefore, we need to ensure that all operations modifying taint information length or content are instrumented. For instance, functions like *append(int i)* change the length of the internal buffer. However, we do not handle those functions in terms of taint propagation, since primitives are not in our scope. Therefore, we only adjust the length of the taint information to keep it in sync with the string length. To perform adjustments, we save the string length at the beginning of a method's execution. Before the method returns, we retrieve the string length of the modified string. By calculating the difference between those

two lengths, we determine the number of characters for adjustment. Then, we increase or decrease the length of the taint information based on the calculated amount.

Primitives. String operations may also take primitives such as `char`, `int` or `byte` as argument. An example is the `append` function of the `StringBuilder` class of the JDK. It allows the programmer to append primitives to an existing string. Since we taint objects, primitives are not included in our tainting approach. Thus, we do not store any taint information about primitives. Without employing the transformation of the whole bytecode, we have two basic options: Either we treat primitives as tainted or untainted. The first option would introduce more false positives. For this reason, we decided to treat all primitives as untainted.

Primitive Arrays. Primitive arrays can be added to strings via certain operations. For instance, the `append` or `insert` functions of the `StringBuilder` class of the JDK. In these functions, we copy the taint information of the array. This copy is then added to the taint information of the string. Nevertheless, elements of primitive arrays can also be accessed directly by index. In this case, we cannot propagate taint information correctly. The reason for that is that we rely on hooking into function calls. On a bytecode level, array element accesses are not performed by a function call. Special instructions exist for every type of primitive array to store and load elements. Propagating taint for primitive arrays would require to intercept all bytecode instructions. Then, they would need to be checked for array instructions to perform further actions. Since we purposely avoid inspection of complete bytecode, this approach is not in our scope. Furthermore, taint propagation for primitive arrays is only reasonable if primitive tainting is supported. For those reasons, we do not propagate taint if parts of a primitive array are stored in a variable.

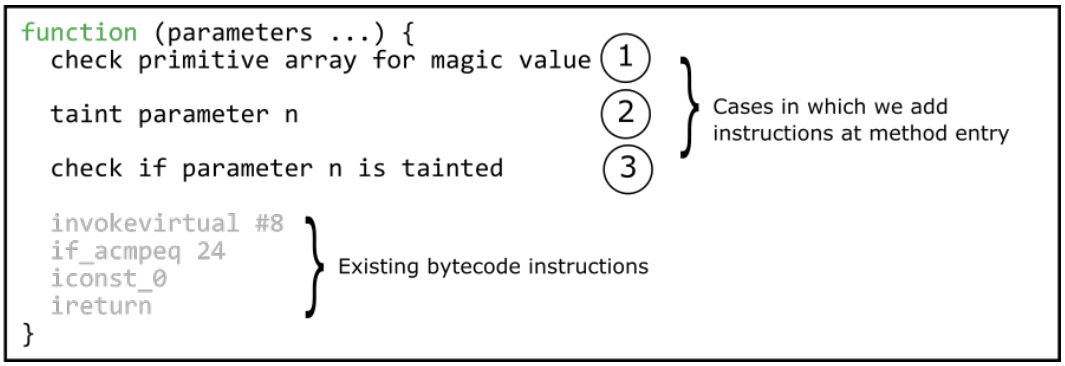

Figure 4.5: Three cases in which we perform instrumentation on method entry.

4.5 Instrumentation

In this section, we explain how our concept uses code instrumentation to implement our taint analysis approach. To stay independent of the JVM language in which the web application is written, we instrument its bytecode. For instrumentation, it must be known when and where to place the instrumentation code. Concerning the point in time, the code can be instrumented offline, during or startup or at runtime. Our approach does not assume a specific time for instrumentation. However, when implementing it for the JVM, a limitation has to be taken into account. This limitation forbids changes to the class-file format after classes are loaded. In other words, functions, fields and implemented interfaces are not allowed to change after a class has been loaded. Since we add an additional field to store taint information, we instrument string classes offline.

Apart from string classes, we do not alter the class-file format of classes. We only add additional instructions to the instruction lists of existing methods. Those additional instructions are added either at the beginning or at the end of the instruction list. In this manner, we can minimise the chance of altering the defined behaviour of existing instructions. The decision whether instructions are added on method entry or exit depends on a method's semantics.

We instrument on method entry when parameters are affected. Method parameters may change during method invocation. Therefore, we place

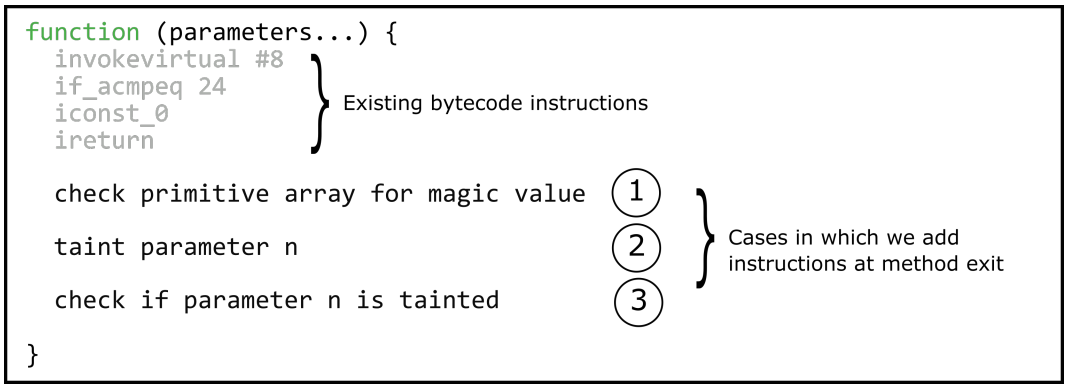

Figure 4.6: Three cases in which we perform instrumentation on method exit.

instrumentation code affecting parameters at method entry. This approach is needed in three cases, as illustrated in Figure 4.5. The first is our interactive tainting approach, in which we compare a method parameter against a magic value. The second case is tainting parameters. By tainting at method entry, we assure that taint information is propagated for the parameter until the end of the method is reached. The last case is to check taint information in sinks. During this check, we inspect security relevant method parameters for taint information.

We instrument on method exit if return values are affected, as variables used as return value may change. Therefore, we place the corresponding instrumentation code affecting them at method exit. We rely on this approach to attach taint information. We perform this step in two cases, namely the initial attachment of taint information and taint propagation. In the first case, we fully taint a return value. In the second case, we copy the taint information of the current instance to the return value.

Length adjustments of taint information represent a special case. Those adjustments are necessary to keep the length of taint information in sync with the string length. Instrumentation on method entry and exit is required for this process. On method entry, the current string length is saved. On method exit, the difference between the stored length and the actual string length is calculated. Based on this difference, adjustments to the taint information length are applied. Details about the implementation of our concept are given in the following chapter.

5 Implementation

This chapter gives details about how we implemented our concept described in Chapter 4. In particular, we show how we used bytecode instrumentation and agents to detect potentially unsafe operations. Our implementation is divided into several components, namely *Runtime Patcher*, *Java Agent*, *Native Agent*. In the following, we describe each of them in greater detail.

Section 5.1 explains how we used ByteBuddy to instrument the web application's bytecode. Section 5.2 describes how we instrument string classes ahead of time with our *Runtime Patcher*. Section 5.3 shows how we use a Java agent to instrument sources, sinks and sanitizers. Section 5.4 illustrates our native agent that we used for attaching taint information to primitive arrays. Section 5.5 covers the results of our conducted evaluation with regard to performance and handling of transformed input.

5.1 Instrumentation Framework

We selected the ByteBuddy framework after comparing it against the well-established frameworks AspectJ, ASM and Javaassist. Details about these frameworks are covered in Section 3.4.3. To implement the taint analysis approach of our concept, we required a framework that supports four main tasks:

1. Insertions of bytecode instructions in the beginning and end of methods instruction list
2. Accessing function arguments in instrumentation code
3. Accessing return values in instrumentation code
4. Matching functions and classes

The first three tasks are supported by all evaluated instrumentation frameworks. The fourth task is only supported out-of-the-box by ByteBuddy and AspectJ. However, without out-of-the-box support of function and method matching, these features need to be reimplemented. Hence, we reduced our possible candidates to ByteBuddy and AspectJ. We finally chose ByteBuddy over AspectJ due to its customisation capabilities. ByteBuddy allows extending the instrumentation functionality at arbitrary locations, such as conditionals.

Listing 5.1: Instrumenting the String class with ByteBuddy.

```
1 final DynamicType.Unloaded<String>
    stringClassUnloaded = new ByteBuddy()
2   .redefine(String.class, classFileLocator)
3   .implement(Taintable.class)
4   .defineProperty(taintInformation, Object.class)
5   .visit(...) // Taint propagation advices
6   ...
7   .make()
```

5.2 Runtime Patcher

In this section, we describe our *Runtime Patcher* component, which instruments string classes ahead of time. We rely on this approach for the `java.lang.String`, `java.lang.StringBuffer` and `java.lang.StringBuilder` classes. First, we show how we add a field which stores the taint information in those classes. Second, we illustrate how we propagate taint information whenever strings are manipulated.

5.2.1 Taint Storage

In order to store taint information, we need to add a place for storage and provide a way to set and retrieve it. Listing 5.1 illustrates how we achieve these tasks with ByteBuddy. We add a field of datatype object which stores the taint information. Furthermore, we also add accessor methods for this

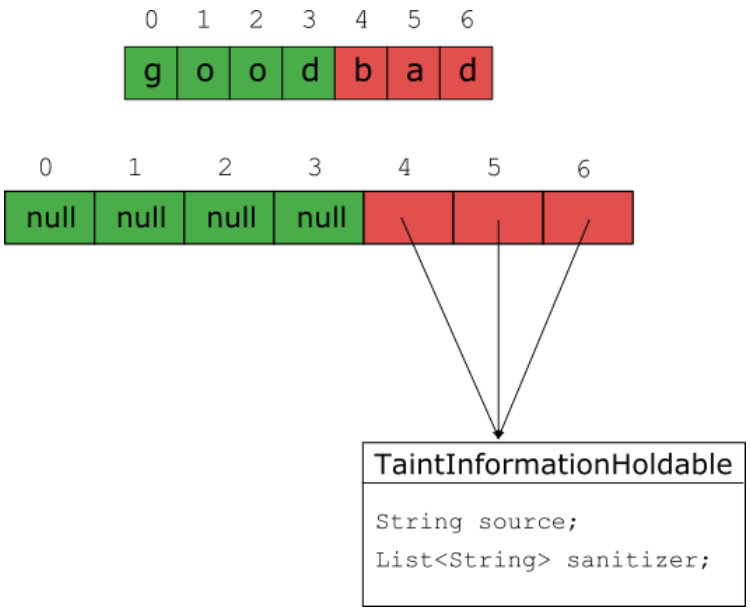

Figure 5.1: Array data structure which stores taint information.

field. Both tasks are accomplished with the `defineProperty` function. In order to call the accessor methods in our instrumentation code, we rely on interfaces and polymorphy. We define the methods in an interface and implement this interface in our string classes. As a result, we can cast our string class to the interface type at runtime and invoke the methods. To accomplish this task, we created the interface `Taintable` and added it via the `implement` function of ByteBuddy. The resulting class of this process is shown in Listing 5.2.

For our prototype, we chose an array data structure for storing taint information, which is illustrated in Figure 5.1. The index of an element corresponds to the position of a character in the string. Each element can either be *null* or hold a reference to an object which contains taint information. This object stores the stack trace of the source function from which the character originated. Furthermore, it holds a list of invoked sanitisation functions.

Listing 5.2: Instrumented `java.lang.String` class.

```
1 public final class String implements Serializable,
    Comparable<String>, CharSequence, Taintable {
```

```
2
3   private final char[] value;
4   private Object taintInformation;
5   public void setTaintInformation(Object var) {...}
6   public Object getTaintInformation() {...}
```

5.2.2 Taint Propagation

Taint propagation constitutes the most substantial part of our work. We perform it for the `String`, `StringBuffer` and `StringBuilder` class of the JDK. In the following, we describe how we propagate taint for those classes. Furthermore, we list which methods we instrumented.

Instrumentation advices for propagation. All propagation code is organised in advices. Each of them fulfils a specific purpose in terms of propagating taint information. Advices marked with an asterisk check for a magic value, as discussed in Section 4.4.

Our first type of advices consists of generic advices. They are used for all string classes.

A1 **Parameter Propagation***
Takes the taint information of the parameter passed to the function and sets in the current instance.

A2 **Return Value Propagation**
Takes the taint information of current instance and sets it to the return value. The return value may be a string class or character/byte array.

A3 **Extract Characters Propagation**
Takes a subset of the taint information of the current instance. This subset is replaced in the taint information of the passed character array at the specified position.

A4 **Substring Propagation**
Takes a subset of the taint information of the current instance and sets it in the returned string.

Furthermore, we required one specific advice for the `java.lang.String` class. This advice is needed for the concatenation operation, since it is different from the one of `java.lang.StringBuffer` and `java.lang.StringBuilder`.

B1 **String Concatenation Propagation**
Concatenates the taint information of the passed string with the taint information of the current string. The new concatenated taint information is assigned to the returned string.

The `java.lang.StringBuffer` and `java.lang.StringBuilder` represent a special case. In addition to advices for propagating the taint information, we needed so-called length adjustment advices. These advices keep the length of the taint information in sync with the string length. This procedure is necessary for all methods in which we do not explicitly handle taint information, but influence the length of it. Omitting this adjustment result in the inability of the JVM to boot.

C1 **Append Propagation**
Appends the taint information of the passed string to the taint information of the current instance.

C2 **Append Length Adjustment**
Increases the length of the taint information by the length of the appended primitives' string representation.

C3 **Insert Propagation***
The taint information of the passed string is inserted at the specified index in the taint information of the current instance.

C4 **Insert Length Adjustment**
Increases the length of the taint information by the length of the inserted primitives' string representation.

C5 **Delete Length Adjustment**
Decreases the length of the taint information by the specified length.

C6 **DeleteAt Length Adjustment**
Delete an element of the taint information at the specified index.

C7 **Replace Propagation**
Takes the taint information of the passed string and replaces the taint information at the specified position in the taint information of the current instance.

C8 **SetCharAt Propagation**
Untaints an element in the taint information at the specified index.
C9 **Reverse Propagation**
Reverses the elements of the taint information.

Advices used by functions of the String class. The String class mostly uses generic advices. This circumstance is owed to the immutability of this class. The majority of functions do not change the content of the string. The following table lists the advices we used to propagate taint information.

Function	**Instrumentation Advice**
Constructor	A1: Parameter Propagation
concat	B1: String Concatenation Propagation
getChars	A3: Extract Characters Propagation
getBytes	A2: Return Value Propagation
replace	Not implemented, see "Further cases." in Section 5.5.4
substring	A4: Substring Propagation
toCharArray	A2: Return Value Propagation
toLowerCase	A2: Return Value Propagation
toString	A2: Return Value Propagation
toUpperCase	A2: Return Value Propagation

Table 5.1: Advices used by the StringBuilder and StringBuffer class.

Advices used by functions of the StringBuffer and StringBuilder class. Since the StringBuffer and StringBuilder class are mutable, we additionally required length adjustment advices. These advices are necessary for all functions which change the string length. Otherwise, the length of the taint information would get out of sync with the string length. The following table lists the advices we used to propagate and adjust taint information.

Function	Instrumentation Advice
append	C1: Append Propagation C2: Append Length Adjustment
Constructor	A1: Parameter Propagation
delete	C5: Delete Length Adjustment
deleteCharAt	C6: DeleteAt Length Adjustment
getChars	A3: Extract Characters Propagation
insert	C3: Insert Propagation C4: Insert Length Adjustment
replace	C7: Replace Propagation
reverse	C8: Reverse Propagation
setCharAt	C8: SetCharAt Propagation
substring	A4: Substring Propagation
toString	A2: Return Value Propagation

Table 5.2: Advices used by the StringBuilder and StringBuffer class.

5.3 Java Agent

In this section, we describe how we implemented our Java agent. In particular, we explain the steps performed until code is instrumented. Furthermore, we elaborate on the tainting and taint checking process in greater detail.

5.3.1 Workflow

The instrumentation performed by our Java agent follows several steps, as illustrated in Figure 5.2. After these steps are completed, all classes which are loaded during bootstrap or application startup are instrumented. Classes loaded after startup are instrumented on class load.

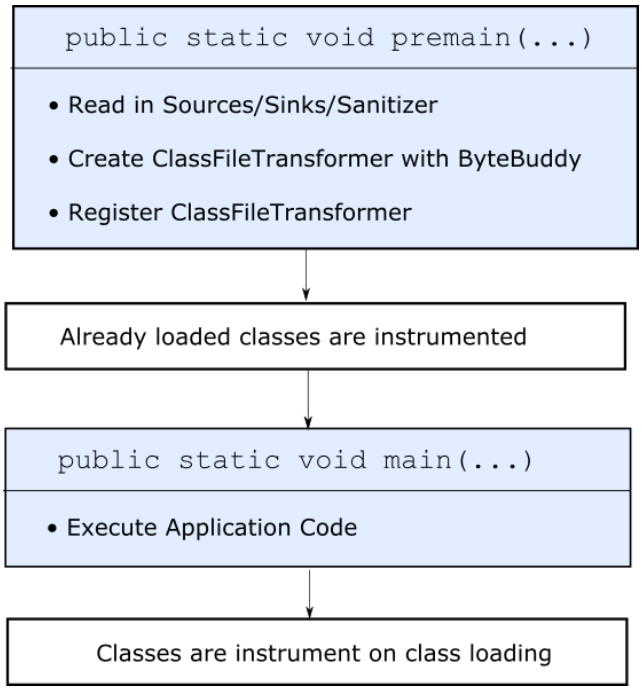

Figure 5.2: Java agent workflow from startup until runtime.

Initially, the `premain` function of the agent is invoked by the JVM. At the beginning of the function, we process the configuration files for sources, sinks and sanitizers. Based on these files, we instruct ByteBuddy to match instrumentation advices to functions. Details about the configuration options are covered in the following section. To perform the instrumentation, a `ClassFileTransformer` instance is required. This transformer object contains class and method matching logic as well as instrumentation code. ByteBudddy creates this object internally, based on how we instruct it.

After the `premain` returns, the Java agent is ready for instrumentation. Subsequently, whenever a class is loaded, it is checked whether methods should be instrumented. Furthermore, if an already loaded class matches, it is instrumented as well. At this stage, only classes required by the Java agent itself are loaded, since the `main` function has not been invoked yet. For those classes, we perform instrumentation to add advices for sources, sinks and sanitizers. Adding taint propagation advices to string classes was not feasible due to restrictions of the ByteBuddy framework. In fact, the ByteBuddy framework could not add the advices because of circularity issues. Therefore, we add those advices ahead of time, as described in Section 5.4.

After the JVM invokes the `main` function, the application is executed. During execution, classes will be loaded when they are used for the first time. Before they are loaded, it is checked whether methods should be instrumented. If this is the case, instrumentation is performed.

5.3.2 Tainting and Taint Checking

The instrumentation code added by our Java agent performs tainting and taint checking. The functions selected for instrumentation are defined in YAML configuration files. They allow matching functions based on several properties. Listing 1 in the Appendix lists an example configuration.

Class matching. First, matching on a class level is possible. Configuration options for class matching are:

- **className** - The name of the class
- **implementedInterfaces** - A list of implemented interfaces
- **superClass** - The super class
- **annotations** - A list of class annotations
- **methods** - A list of methods to instrument, further described in the following paragraph.

Method matching. After the class is identified, the listed methods are matched. Configuration options for method matching are:

- **name** - The name of method
- **descriptor** - A method descriptor, as defined in the JVM specification [LB15, Section 4.3.3]
- **advice** - The advice containing the instrumentation code
- **parameterToTaint** - A list of parameters to taint (only for sources)
- **taintReturnValue** - Whether to taint the method return value (only for sources)
- **parametersToCheck** - A list of method annotations (only for sinks)

Matcher transformation. To identify classes and functions based on the above-mentioned properties, we transform them into matchers suitable for ByteBuddy. These matchers perform the actual matching process on a bytecode level. After a method is successfully identified, the code in the instrumentation advice is added.

Tainting. For sources, our advice code sets the taint information. Since we store taint information per character, we create a new array with the same length as the string itself. For string classes, the taint information is then set via a setter method added to the string classes by our *Runtime Patcher* component. For character and byte arrays, the taint information is set via a native method of our *Native Agent*. In both cases, taint information in the form of an array is attached.

Our advice for sanitisation functions records if a string, passed as parameter, has been sanitised. Whenever sanitisation methods return, we store in the

taint information of the return value that the function has been invoked. Since a string might be sanitised by more than one sanitisation function, we support recording several ones.

Taint checking. Taint information checking is performed in advices for sinks. By default, our sink checking advice is applied, which logs vulnerabilities when input is partially tainted. To perform a more detailed analysis, custom advice classes can be configured. In doing so, one can take into account tracked sanitisation functions to reduce false positives. Furthermore, advices can be parameterised by passed function parameter indices. For instance, let us consider a function which takes two parameters and executes a SQL query. The first parameter represents the actual query string. The second parameter contains prepared arguments that cannot lead to an injection. In this example, the configuration contains only a index entry for the first parameter.

5.4 Native Agent

Our native agent provides functions used by our tainting advices to attach taint information to primitive arrays. In particular, we implemented functions to set and retrieve tags for arbitrary objects. These tags are linked to objects which store taint information. We implemented two native functions for that purpose, namely `_setTag` and `_getTag`. However, they need to be manually registered, in order to use them in our Java agent code. Therefore, we have to list them as a prototype, since the native agent implementation is in is C++ code. Listing 5.3 shows how we register the `_setTag` function. We further introduce a wrapper function `tagObject`. In doing so, we can check whether the native agent is loaded before calling the native function.

Listing 5.3: Registration and usage of the _setTag native function.

```
1 private static native void _setTag(Object obj,
    Object t);
2
3 public static void tagObject(Object obj, Object t) {
4   // check if native agent is loaded
```

```
5     _setTag(obj, t);
6 }
```

To implement tagging capabilities in our `_setTag` and `_getTag` native functions, we use the `SetTag` and `GetTag` function provided by the JVMTI. In order to use these JVMTI functions, we register the *can_tag_objects* capability. However, the `SetTag` cannot be used directly to attach an object, as it only supports tags of datatype long. Therefore, we create a new global reference to the object we want to associate and cast this reference to a long, as illustrated in Listing 5.4. In this way, we can associate object tags to objects.

Listing 5.4: Tagging `obj` with a pointer to `taintInformationObject`.

```
1 jvmti->SetTag(obj, (jlong) (ptrdiff_t) (void*)
    env->NewGlobalRef(taintInformationObject));
```

5.5 Evaluation

In this section, we evaluate our implementation in terms of handling of manipulated inputs and performance. Concerning manipulated inputs, we analyse effects on taint information based on several categories of operations, such as string functions returning strings or string functions returning primitives. With regard to performance, we analyse the impact on application startup and request handling. Overhead on startup duration is caused by matching and instrumenting sources, sinks and sanitisers. Applications may grow over time and so the number of classes and methods. Furthermore, the number of matchers in the configuration files might increase as well. All those parameters have an impact on startup time. Hence, we analyse the influence of their growth. Request handling overhead is caused by propagating the taint information. As dynamic scanner might send a high number of requests to the application, throughput must be high enough to allow efficient scanning. Therefore, we measure the overhead on response times and throughput.

To evaluate the performance indicators startup time, response time and throughput, we need to choose finite bounds for parameters. For instance,

a specific limit on the number of loaded classes or the payload size. We chose those bounds to be able to infer trends concerning runtime behaviour. They do not imply a particular limit of our implementation. Furthermore, when evaluating a particular performance indicator, we also have to choose fixed parameters. Examples for those include a particular amount of classes, methods or matcher. Again, we chose those in a way that allows us to infer trends in runtime behaviour.

5.5.1 Setup

We perform our evaluation on a Spring boot application. To measure startup time, we enhanced it to load a specific number of classes. To assess our detection capabilities, we added deliberately vulnerable endpoints. Other insecure open-source applications exist, such as OWASP WebGoat[1] or JavaVulnerableLab[2]. However, those applications do not focus on test cases for blind injections that include input manipulations. Since we require test cases for that purpose to assess our detection capabilities, we developed a custom test set. This test set focuses on the effects of input manipulations on IAST detection capabilities.

Startup overhead tests are performed on a 2.1GHz Intel Xeon Silver 4116 CPU / 16 GB RAM machine. Request overhead is evaluated on 2.7GHz Intel Core i7-6820HQ CPU / 16 GB RAM machine. As Java runtime, we use JDK version 1.8.0_192.

Used dependencies We rely on several well-established spring projects. Furthermore, we used relational and document-based databases to test SQL and NoSQL injections. For this purpose, we used in-memory databases. The *h2* database serves as our relational database. As document-based database, we use an embedded version of *mongoDb*.

[1] https://github.com/WebGoat/WebGoat
[2] https://github.com/CSPF-Founder/JavaVulnerableLab

Dependency	Version
org.springframework.boot:spring-boot-starter-web	2.0.4.RELEASE
org.springframework.boot:spring-boot-starter-data-mongodb	2.0.4.RELEASE
org.springframework:spring-jdbc	2.0.4.RELEASE
cz.jirutka.spring:embedmongo-spring	1.3.1
org.owasp.encoder:encoder	1.2.2
com.h2database	1.4.197

Table 5.3: Dependencies used in our vulnerable web application.

5.5.2 Overhead on Application Startup

In this section, we evaluate the performance impact on application startup in terms of duration. The impact is mainly caused by matching sources, sinks and sanitisation functions. As described in Section 5.3.2, we support matching based on different properties. In this evaluation, we set our focus on the properties class name and method name. Since they might cause different overheads, we evaluate them separately. To prevent falsification of measurements, we vary only one of the following parameters at a time: The number of classes, the number of methods or the number of matchers.

Measurement Procedure. To measure the overhead, we perform the following procedure: First, we generate a particular number of classes. Depending on our scenario, we add several methods to these classes. Furthermore, we may generate several matchers. Next, we compile our application with the generated classes and start it. On startup, we load each of the generated classes in turn. After the application booted, we measure the startup time of the JVM using the `ManagementFactory.getRuntimeMXBean().getUptime()` method. To get a reasonable amount of samples, we start the application 35 times in sequence. The data points in the following graphs are calculated by taking the median of those 35 measurements. In doing so, we reduce the effects of outliers.

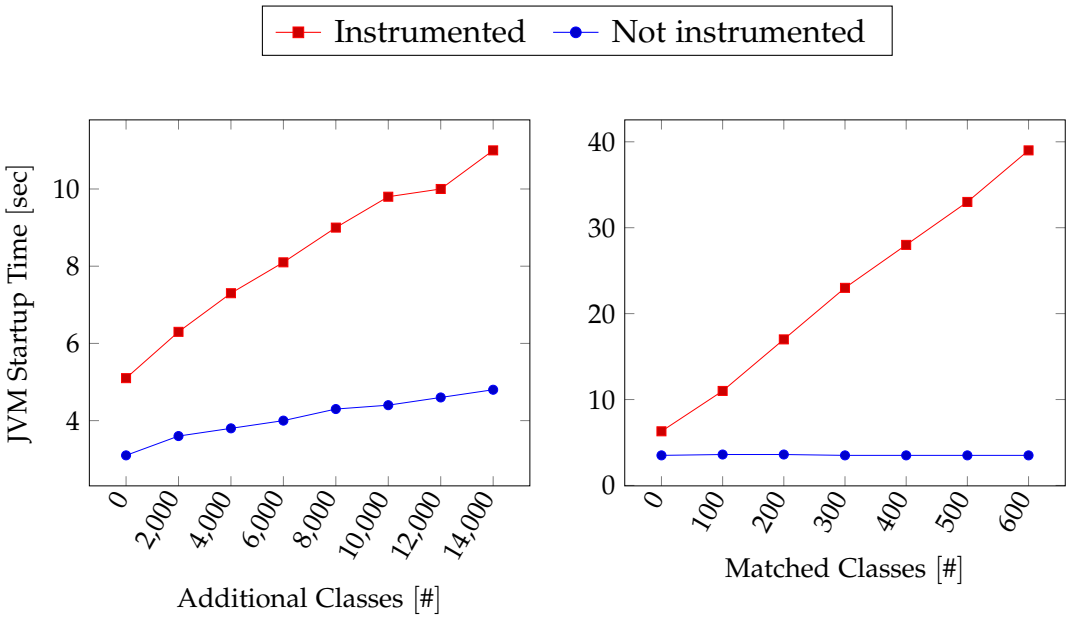

Figure 5.3: The first plot shows the increase in startup time depending on the number of additionally loaded classes.
The second plot depicts the increase in startup time depending on the number of matched classes (using 2000 additional classes).

Class and Method Name Matching. We test the scalability of class and method name matching according to two parameters. The first parameter is the number of elements to match, which are either classes or methods. We perform measurements by specifying exactly one name matcher and increasing the number of elements. The second parameter is the number of name matchers in a configuration file. In this case, we measure by increasing their amount at a fixed number of elements to match. Since the name comparison takes constant time with respect to our tested parameters, we expect an approximately linear increase of startup time. We illustrate our measurements for class name matching in Figure 5.3. The behaviour of method name matching is depicted in Figure 5.4. As a baseline, we take measurements without applying our agents. Since no matching is performed, approximately constant behaviour is observable. For the instrumented application, we observe linear increase of start time, confirming your hypothesis.

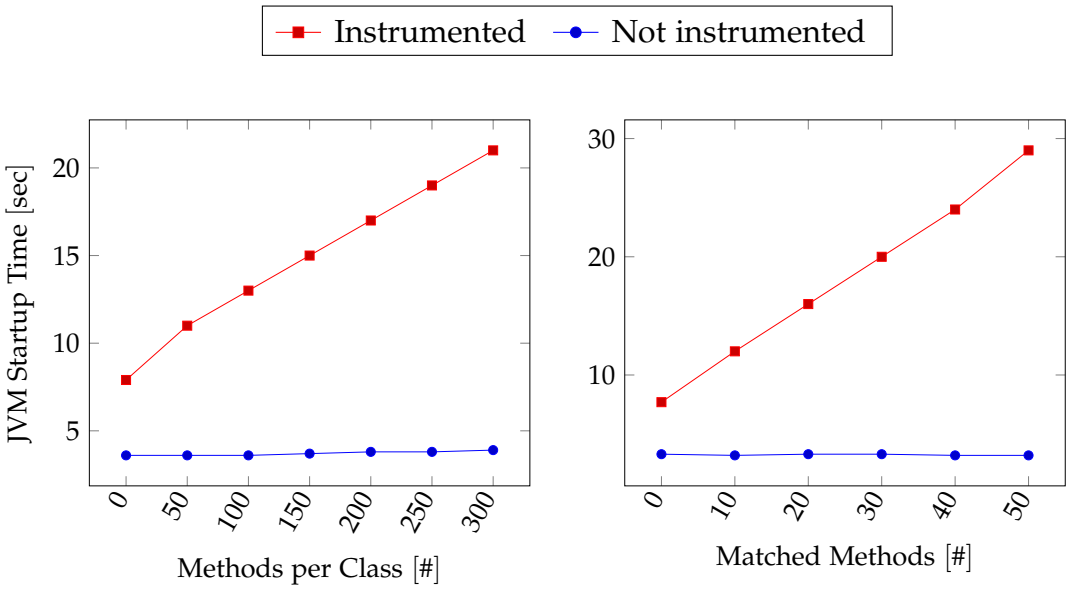

Figure 5.4: The first plot shows the increase in startup time depending on the number of methods in a class (using 2000 additional classes).
The second plot illustrates the increase in startup time depending on the number of matched methods (using 500 additional classes and 100 methods per class).

Method Instrumentation. We test the scaling of the instrumentation process according to the number of methods to instrument. The instrumentation process is performed in constant time with respect to the number of instrumented methods. Hence, we expect a linear increase in startup time. The resulting graph in Figure 5.5 confirms our hypothesis of linear behaviour.

5.5.3 Overhead on Request Handling

In this section, we evaluate the performance impact on HTTP request handling. For this purpose, we measure the Time To Last Byte (TTLB) in milliseconds and the throughput in requests per seconds. The impact is caused by taint propagation algorithms which copy and manipulate taint information. Those algorithms differ for every instrumented string operation listed in Section 5.2.2 Therefore, we evaluate each operation separately. As sink, we use the `query` function of the `org.springframework.jdbc.core.JdbcTemplate`

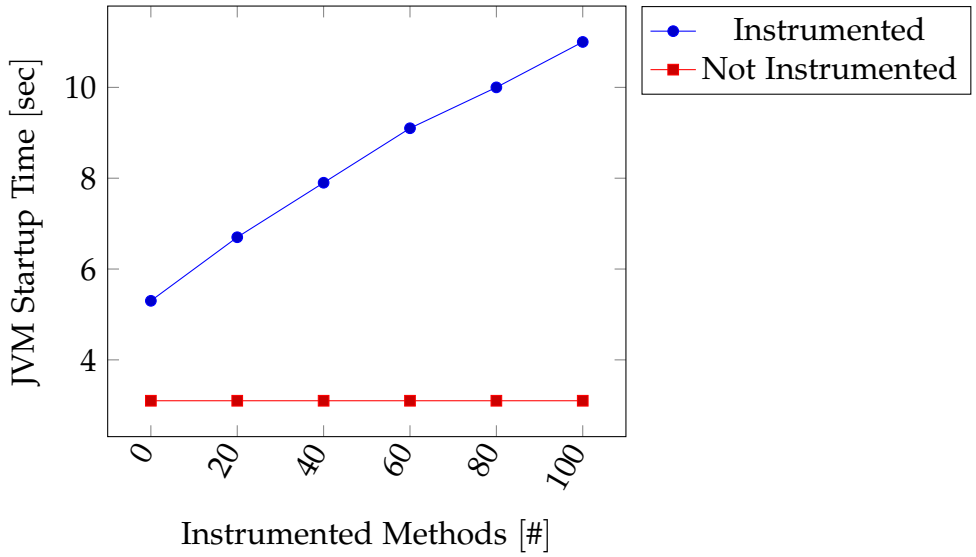

Figure 5.5: Increase in startup time depending on the number of instrumented methods (in a single class with 800 methods)

class. To perform our measurements, we use JMeter[3], an industry standard tool for measuring performance impact. Furthermore, we take different payload sizes into account, since the overhead may depend on these.

We measure according to the following procedure: For each operation and payload size, we send 400 requests to acquire a reasonable amount of samples. Next, we aggregate our results by taking the median of the response times and throughputs for every <*operation*, *payload size*> tuple. We chose the median as we observed outliers due to garbage collection and runtime code optimisation. Finally, we group the tuples by payload size and calculate the average response time and throughput for each group. The resulting dependency between response time, throughput and payload size is plotted in Figure 5.6.

As observable in Figure 5.6, increasing the payload size results in an approximately linear increase of the response time. The overhead for all payload sizes is on average 35%. Throughput, on the other hand, decreases indirect

[3] https://jmeter.apache.org/

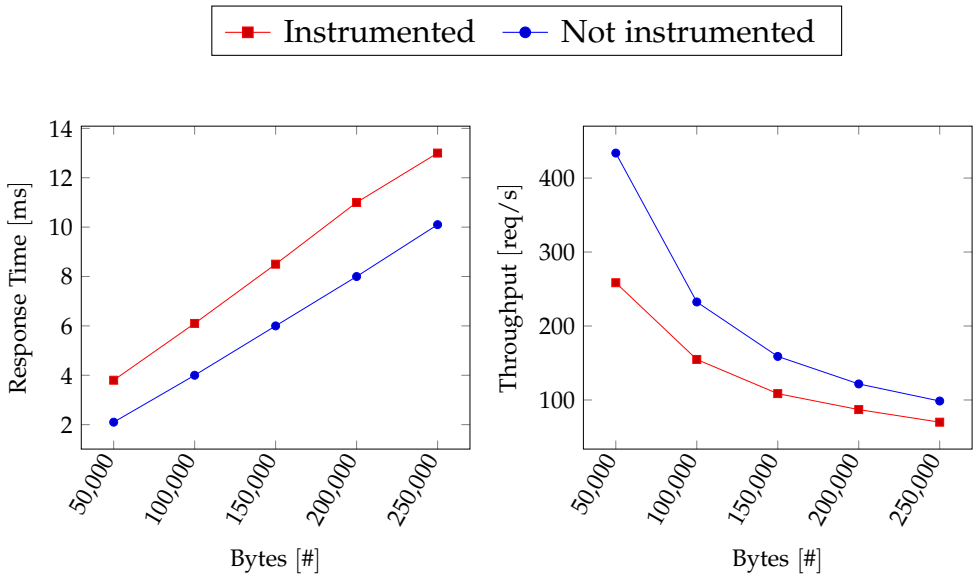

Figure 5.6: Response time and throughput depending on the payload size.

proportional. It is a functional equation in the form $\frac{1000\ [ms]}{average\ response\ time\ [ms]}$. As a result, we observe a hyperbolic curve.

5.5.4 Effect of Input Manipulations

Input manipulations are the main cause limiting the detection capabilities of IAST. We assess the influence of these manipulations based on several categories. These categories represent operations that may be performed on tainted strings before they reach a sink. For each of these categories, we test its influence on taint information.

String functions taking/returning strings. This category includes all functions of the string classes, which take byte/character arrays or string class objects as a parameter. Section 5.2.2 lists those functions and gives further details on how they are instrumented. We developed a test suite to test this category. It contains functional tests for all string functions we

instrumented. For instance, we assure whether a call to the `reverse` function of the StringBuilder class reverses the taint information. Analogously, we ensure for other methods that its semantics is correctly reflected on the taint information. If strings are passed to functions which internally use functions of this category, taint information is propagated as well. For instance, the `java.lang.StringTokenizer` class uses to substring functions. If one decides to split a string by using such a utility class, taint information is propagated accordingly.

String functions returning primitives. This category includes all operations that extract primitives from a string. Two functions are included in our approach, all of which are defined in every string class we instrument. The `codePointAt()` functions returns a unicode character in form of an integer. The `charAt()` returns a one byte character. Since both return values are primitive types, we do not attach taint information to them. Consequently, these values become untainted.

String functions taking primitives. This category includes string operations which add primitives to a string. These are the `append` and `insert` function of the StringBuffer and StringBuilder class. In both functions, the string representation of a primitive is added to an existing string. Since we treat primitives as untainted in our approach, the added characters will be untainted as well. Therefore, loss of taint information occurs when the added characters are originally tainted. A potentially dangerous case is if several characters are extracted into primitives and then added again with the `append` or `insert` function. For a non-adversarial developer, we hypothesise that this occurrence is rare.

Tainted Array accesses. This category includes read and write operations on a tainted primitive array. If an element is read, taint information is not propagated. Analogously to the category *String functions returning primitives*, elements get untainted. If an element is overridden, it remains tainted. This circumstance can lead to false positives, since transformed parts of an array may be copied and used in a sink. However, false positives can be reduced

by encapsulating transformations in a method and registering this method as a sanitiser. For instance, let us consider the `java.util.Base64.encode()` function that transforms an entire primitive byte array. If a sink is safe with Base64 encoded input, one could define the `Base64.encode()` method as a sanitiser. By checking for this sanitiser in a sink, a potential false positive can be suppressed.

Furthermore, it is possible to use the `System.arrayCopy()` to copy chunks or entire arrays. In our current demonstrator, the array copy will not inherit the taint information. Instrumenting the `System.arrayCopy()` function represents future work. Manual copies represent a special case. If the copy is performed without any manipulations to the individual elements, taint information is not propagated. However, in this case, the magic value can be checked in the sink to prevent false negatives. The worst case represents array copies in which only a subset of elements is transformed. If such a method is not a sanitisation method, false negative arise. For a non-adversarial developer, we hypothesise that this occurrence is rare.

Further cases. In this category, we list further relevant cases in which we do not propagate taint information. Those cases are not implemented and represent future work.

When serialising String instances, the taint information is not serialised. Although these instances are objects, internally they are serialised differently. More specifically, the `writeString` method is used instead of the `writeOrdinaryObject` method. StringBuffer and StringBuilder instances are serialised using the `writeOrdinaryObject` method. Therefore, taint information is serialised. In order for String instances to be serialised as well, the `writeString` method needs to be modified. For primitive arrays, the `writeArray` needs to be modified.

The `String.format` function supports constructing a string by specifying a format string with placeholders. The placeholder arguments to the format string may be tainted. To propagate taint information accordingly, the `java.util.Formatter` class needs to be instrumented.

Regex replacements are performed using the `java.util.regex.Pattern` and `java.util.regex.Matcher` classes. Propagation of taint information

requires instrumenting of those classes. In doing so, the `replace` method of the String class could also propagate taint information.

6 Related Work

In this chapter, we cover related work of security testing. As Interactive Application Security Testing (IAST) can incorporate both static and dynamic testing, we cover work of both methodologies. Furthermore, we list approaches for dynamic taint analysis of JVM applications.

Static Analysis In the following, we list different static analysis approaches. Our approach differs from these since we test an application in its running state. Therefore, false positives resulting from the over-approximation are reduced, as pointed out by Trinh et al. [TCJ14].

Thomé et al. [Tho+18] use a form of slicing to improve security auditing. In slicing, code slices are extracted according to specific criteria. One such criterion is, for instance, all program statements leading to a specific sink. By slicing according to this criterion, they calculate slices for security auditing. With such slices, manual auditing is more effective. Only relevant statements for a specific sink need to be audited.

Livshits and Lam [LL05] use points-to analysis for solving the tainted object propagation problem. Points-to analysis uses allocation sites, which represent an object of a specific type. The result is points-to relations between variables and allocations sites. They detect potential security violations by checking if a sequence of points-to relations leading from a source to a sink exists.

Taint analysis, a kind of information flow analysis, is another concept for solving the tainted object propagation problem. Although points-to analysis and taint analysis are seen as different processes, they can be unified. Grech and Smaragdakis [GS17] propose a unification which uses points-to algorithms to implement information-flow analysis. In taint analysis, each

variable is assigned a particular taint status to mark it as user input. This approach can be implemented with several techniques. Cao et al. [Cao+17] apply taint analysis by utilising a Control Flow Graph (CFG) build out of an AST. Tripp et al. [Tri+09] use code slicing techniques for that purpose.

Chaudhuri and Foster [CF10] employ symbolic execution to detect vulnerabilities such as session manipulation and unauthorised access. In symbolic execution, variables are represented symbolically. Conditions represent constraints on symbolic variables. Using simple assume/assert language, security properties are defined. If satisfiable paths constraints leading to assertion failures are found, a security violation is detected. In order to solve these constraints, they use the Yices Satisfiability Module Theory (SMT) solver Yices[1].

A further application of symbolic execution is string constraint solving. Trinh et al. [TCJ14] developed such a solver based on the Z3[2] SMT solver. Validation routines in an application can block potentially malicious input. However, these routines may not entirely block malicious inputs. String constraint solver can be used to assess whether this is possible and thus detect more vulnerabilities.

Monshizadeh et al. [MNV14] proposed an approach which detects privilege escalation. They check if the same authorisation context is used consistently for the same resource across different paths within the application. In case of inconsistencies, the access control policy may be violated.

Near and Jackson [NJ14] introduced a solution for missing security checks. Using symbolic execution, they interactively construct a security policy by questioning the user about ways in which data might be exposed. With this policy, they can detect missing security checks.

Dynamic Analysis Subsequently, we list different approaches which dynamic analyser may use. These approaches differ, as they do not apply dynamic taint analysis.

[1] http://yices.csl.sri.com/
[2] https://github.com/Z3Prover/z3

Awang and Manaf [AM15] proposed a solution which generates test cases based on known attack patterns. Injection attack patterns, such as tautologies or illegal or queries. They use the generated test cases to build URLs and target parameters to construct requests. The responses to those requests are analysed to decide whether an attack was successful. In particular, they seek for error messages or authentification bypass.

A black-box approach for improving coverage was proposed [Dou+12]. First, a model of the internal state is built incrementally. Based on this model, a state machine is built. This state machine is used to drive an open-source fuzzing tool.

Another black-box methodology to improve coverage is evolutionary fuzzing. Duchene et al. [Duc+14] performed this technique by using a genetic algorithm to adapt to sanitisation functions. A grey-box approach was proposed by Liu Qiang et al. [LLW14]. It leverages Web Service Description Language (WSDL) definitions to generate abnormal data for fuzzing efficiently. Nevertheless, neither black- nor grey-box fuzz testing knows any internal constraints. To address this limitation, combinations of fuzzing with symbolic execution [GLM08] were proposed. However, this solution was not tested if it integrates with web applications.

Khoury et al. [Kho+11] tested scanners from the vendors Acunetix, IBM and QualysGuard. They implemented a custom testbed with stored SQL Injections (SQLIs) vulnerabilities. At that point in time, the tested scanners were not able to detect these vulnerabilities. Even when they instructed the scanners on how to exploit the vulnerabilities, none were detected.

Dukes et al. [DYA13] tested tools, including Paros, WebScarab, JBroFuzz, Acunetix and Fortify. Furthermore, they performed manual testing as well. They conclude that some types of vulnerabilities are not found with automated tools. Therefore, manual testing is still an important part of security testing. Using a variety of tools together with manual testing finds the most number of vulnerabilities.

A more recent study [PZK16] showed that stored injections still impose a problem. They evaluated the detection capabilities of Acunetix, AppScan and ZAP with regard to SQL injection and Cross-Site Scripting (XSS). Their results suggest that choosing the right attack vectors is crucial. Furthermore,

they highlight that stored SQL injection and XSS are especially challenging to detect. No scanner was able to detect a stored SQL injection vulnerability created by them.

Further studies [HN17; Alz+17] indicate the limitations of automatic scanning. Holik and Neradova [HN17] point out that modern web technologies have an impact on the detection capabilities. Most of the undiscovered vulnerabilities were caused because of these technologies. Therefore, they suggest using automated tools to get a general idea about the application security status. For comprehensive security testing, they recommend more specialised tools, scripting and manual testing.

Dynamic Taint Analysis In the following, we list dynamic taint analysis approaches for the JVM. Dynamic Taint Analysis (DTA) is a well-known technique which was proposed by Newsome and Song [NS05]. It is a form of information-flow analysis (IFA). The basic idea is to tag data from untrusted sources as tainted. Instrumentation code propagates these tags until they reach a potentially vulnerable function, a so-called sink. Tagging and tag propagation are carried out by source or binary code instrumentation of the target application.

Masri et al. [MPL04] proposed techniques for Dynamic Information Flow Analysis (DIFA) on the JVM platform. In order to find information flows from sources to sinks, they apply dynamic slicing. The information about the data flow for dynamic slicing is collected by profiling code. This code is added by bytecode instrumentation. Moreover, they use an optional static preprocessing phase to identify information flows. Dynamic analysis is inherently unable to detect these implicit information flows. Their approach can be applied statically for validation or dynamically to prevent illegal information flows.

An approach for DTA was applied by Haldar et al. [HCF05]. Like in the approach presented by Masri et al. [MPL04], sources and sinks have to be specified. Strings originating from untrusted sources are marked as tainted using a boolean flag. This flag is added to the Java String, StringBuffer and StringBuilder class. Furthermore, all methods in the String class, which modify a string passed as a parameter, propagate the taint flag to the

resulting string. An example of such a method is the *toLowerCase* method. However, the authors do not take primitives and arrays of primitives into account. This circumstance may lead to loss of taint information. One such scenario is the conversion of a tainted string to an array of primitive characters. Another limitation of their approach is that taint-information is not stored at character-granularity. As a result, fine-grained analysis of tainted strings is not possible.

A similar but improved approach was suggested by Chess and West [CW08]. In addition to the taint status of a string, it also stores information about the source of tainted data. This information allows the categorisation of vulnerabilities. For example, a tainted string originating from a local file may have a different severity level than that of an HTTP parameter. Furthermore, they use load-time instrumentation instead of compile-time instrumentation.

As an improvement for tainting at string granularity, approaches for character-granular tainting [HOM06; CW09] have been proposed. In these approaches, the taint status is stored for every single character in a string. Knowing which parts of a string are tainted may help in reducing false positives. The reason for that is, according to them, that strings may contain benign taint information. A limitation of the proposed approaches is again that they do not propagate taint information among primitives. As a consequence, storing a string in a primitive character array results in the loss of taint information. We encounter this limitation by using the object tagging capabilities of the JVMTI in a native agent. Furthermore, we do not store taint in a boolean array. We use references to objects which can store more meta-data such as source and list of applied sanitizers. Finally, we also reduced the dependency on a comprehensive definition of sources with our interactive tainting approach.

To address the tainting of primitives, Bell and Kaiser [BK15] developed a new approach. It propagates taint information for primitives and arrays of primitives. In particular, the taint status is stored in the form of a shadow variable or shadow array. For a primitive variable, a shadow variable is stored in an adjacent location on the stack. Analogously, a shadow array is stored adjacent to each array of primitives. The current status of their work is published on GitHub[3]. Despite their innovative handling of primitives,

[3] https://github.com/gmu-swe/phosphor

their implementation is not designed to be non-invasive. Since they add instructions between existing bytecode instructions, it is more likely that invalid bytecode is generated. For instance, we observed that their implementation did not integrate with embedded web applications container. In contrast, our approach is less invasive by adding new instructions only at method entry or exit. Hence, our solution integrates also with embedded web application containers.

7 Conclusion

The rise of Industrial Internet of Things (IIoT) cloud platforms introduced a myriad of services, typically powered by web applications. These applications process potentially sensitive data send from IIoT devices, such as domain knowledge and intellectual property. However, vulnerabilities in web applications may allow an attacker to access or manipulate sensitive data. Thus, vulnerabilities in these applications impose a potential threat to the industry. In many instances, vulnerabilities are caused by operations which can be adversely used when invoked with user input, and thus are potentially unsafe. Interactive Application Security Testing (IAST) is an approach to detect such operations. A typical IAST approach is to include a particular value in requests. This value is checked in potentially unsafe operations. However, input manipulations can manipulate the value and thus introduce false negatives. Consequently, IAST needs to counter such input manipulations in order not to miss vulnerabilities. To allow fine-grained analysis of manipulations, tracking at character level is crucial. Otherwise, false positives arise because of over-approximation.

In this thesis, we developed an IAST solution that encounters input manipulations. To achieve this goal, we combined the aforementioned magic-value based approach with dynamic taint analysis at character-level granularity. For the implementation of our approach, we relied on Java bytecode instrumentation. To minimise interferences with the application code, we only added new bytecode instructions. First, we instrumented string-holding classes pre-runtime. We added an additional field to store taint information. Furthermore, we also instrumented string-manipulating functions to propagate this taint information. This propagation primarily consists of mimicking the performed manipulations on the taint information. The attachment of taint information in sources, as well as the taint checking, is

performed on-demand using a Java agent. We further reduced the dependency on a comprehensive definition of sources by an approach which we refer to as *interactive tainting*. Hereby, we use the magic value to taint strings automatically which are created from it.

Although taint propagation for string classes covers a broad spectrum of manipulations, character or byte arrays may be extracted from string classes. These array classes are not covered by our instrumentation approach for string classes, since arrays cannot be instrumented without modifying the underlying Java Virtual Machine (JVM). Nevertheless, if such an array is passed to a sink, a potential vulnerability is missed. To have the possibility to attach taint information to them, we implemented a native agent. By using object tagging capabilities JVM Tool Interface (JVMTI), we can attach taint information to character and byte arrays.

The results of our evaluation suggest that combining IAST with taint analysis for web applications is practical. Nevertheless, our approach adds performance overhead. In the context of a Spring boot application, we observed an average overhead in HTTP response time of 35%.

With our implementation, we demonstrate the practicality of our approach. We implemented an IAST solution that reduces the effects of input manipulations. By incorporating dynamic taint analysis, we showed that IAST detection capabilities for JVM web applications can be improved. Consequently, we propose other IAST solution to incorporate taint analysis. Thereby, more vulnerabilities are detected, and ultimately potential damage to the industry is prevented.

Appendix

Listing 1: Sources configuration file.

```
1 className: 'org.example.Test'
2 implementedInterfaces: ['com.example.Interface1',
    'com.example.Interface2']
3 superClass: 'com.example.SuperClass'
4 annotations: ['com.example.Annotation1',
    'com.example.Annotation2']
5 methods:
6 - name: 'methodName'
7  descriptor:
     '(Ljava/lang/String;)Ljava/lang/String;'
8  parameterToTaint: [1,2,3]
9  taintReturnValue: true
```

Bibliography

[Alz+17] A. Alzahrani et al. "Web Application Security Tools Analysis." In: *2017 IEEE 3rd International Conference on Big Data Security on Cloud (BigDataSecurity), IEEE International Conference on High Performance and Smart Computing, (HPSC) and IEEE International Conference on Intelligent Data and Security (IDS)* (2017), pp. 237–242. ISSN: 2325-8071. DOI: 10.1109/BigDataSecurity.2017.47. (Cit. on p. 72).

[AM15] N. F. Awang and A. A. Manaf. "Automated Security Testing Framework for Detecting SQL Injection Vulnerability in Web Application." In: *Global Security, Safety and Sustainability: Tomorrow's Challenges of Cyber Security*. Ed. by H. Jahankhani et al. Cham: Springer International Publishing, 2015, pp. 160–171. ISBN: 978-3-319-23276-8 (cit. on p. 71).

[BEN19] O. Ben-Kiki, C. Evans, and I. döt Net. *YAML*. 2019. URL: https://yaml.org/spec/1.2/spec.html (visited on 03/17/2019) (cit. on p. 42).

[BK15] J. Bell and G. Kaiser. "Dynamic taint tracking for Java with phosphor (demo)." In: *Proceedings of the 2015 International Symposium on Software Testing and Analysis - ISSTA 2015* (2015), pp. 409–413. DOI: 10.1145/2771783.2784768. (Cit. on p. 73).

[BLC02] E. Bruneton, R. Lenglet, and T. Coupaye. "ASM: A Code Manipulation Tool to Implement Adaptable Systems." In: *Adaptable and Extensible Component Systems* (2002). DOI: 10.1.1.117.5769 (cit. on p. 28).

[Cao+17] K. Cao et al. "PHP Vulnerability Detection Based on Taint Analysis." In: *Cao, Kai, et al. "PHP vulnerability detection based on taint analysis." 2017 6th International Conference on Reliability, Infocom Technologies and Optimization (Trends and Future Directions)(ICRITO)* (2017), pp. 4–7 (cit. on pp. 10, 70).

[CB] S. W. Chan and E. Burns. *Java Servlet Specification*. URL: https://jcp.org/aboutJava/communityprocess/final/jsr369/index.html (visited on 08/30/2018) (cit. on p. 17).

[CF10] A. Chaudhuri and J. S. Foster. "Symbolic security analysis of ruby-on-rails web applications." In: *Proceedings of the 17th ACM conference on Computer and communications security - CCS '10* (2010), p. 585. ISSN: 15437221. DOI: 10.1145/1866307.1866373 (cit. on pp. 11, 70).

[Chi00] S. Chiba. "Load-time structural reflection in Java." In: *European Conference on Object-Oriented Programming* (2000), pp. 313–336. ISSN: 0302-9743. DOI: 10.1007/3-540-45102-1_16 (cit. on p. 28).

[CW08] B. Chess and J. West. "Dynamic taint propagation: Finding vulnerabilities without attacking." In: *Information Security Technical Report* 13.1 (2008), pp. 33–39. ISSN: 13634127. DOI: 10.1016/j.istr.2008.02.003 (cit. on p. 73).

[CW09] E. Chin and D. Wagner. "Efficient Character-level Taint Tracking for Java." In: *Proceedings of the 2009 ACM Workshop on Secure Web Services*. SWS '09. Chicago, Illinois, USA: ACM, 2009, pp. 3–12. ISBN: 978-1-60558-789-9. DOI: 10.1145/1655121.1655125. (Cit. on p. 73).

[Dou+12] A. Doupé et al. "Enemy of the State: A State-Aware Black-Box Web Vulnerability Scanner." In: *USENIX Security Symposium* (2012), pp. 523–538. ISSN: 0219-1377. DOI: 10.1007/BF03325089. (Cit. on pp. 13, 71).

[DS] L. DeMichiel and B. Shannon. *Java EE Specification*. URL: https://jcp.org/aboutJava/communityprocess/final/jsr366/index.html (visited on 08/30/2018) (cit. on p. 19).

[Duc+14] F. Duchene et al. "KameleonFuzz: evolutionary fuzzing for black-box XSS detection." In: *Proceedings of the 4th ACM conference on Data and application security and privacy*. ACM. 2014, pp. 37–48 (cit. on pp. 13, 71).

[DYA13] L. Dukes, X. Yuan, and F. Akowuah. "A case study on web application security testing with tools and manual testing." In: *2013 Proceedings of IEEE Southeastcon* (2013), pp. 1–6. ISSN: 07347502. DOI: 10.1109/SECON.2013.6567420. (Cit. on p. 71).

[Erno3] M. D. Ernst. "Static and dynamic analysis: synergy and duality." In: *WODA 2003 ICSE Workshop on Dynamic Analysis* (2003), pp. 24–27. ISSN: 0270-5257. DOI: 10.1.1.2.3628. (Cit. on p. 13).

[Gara] Gartner. *Dynamic Application Security Testing (DAST)*. URL: https://www.gartner.com/it-glossary/dynamic-application-security-testing-dast/ (visited on 04/11/2019) (cit. on pp. 2, 10).

[Garb] Gartner. *Interactive Application Security Testing (IAST)*. URL: https://blogs.gartner.com/neil_macdonald/2012/01/30/interactive-application-security-testing/ (visited on 04/11/2019) (cit. on pp. 3, 10).

[Garc] Gartner. *Static Application Security Testing (SAST)*. URL: https://www.gartner.com/it-glossary/static-application-security-testing-sast (visited on 04/11/2019) (cit. on pp. 2, 10).

[GGS14] M. K. Gupta, M. C. Govil, and G. Singh. "Static analysis approaches to detect SQL injection and cross site scripting vulnerabilities in web applications: A survey." In: *International Conference on Recent Advances and Innovations in Engineering, ICRAIE 2014* (2014), pp. 9–13. ISSN: ' DOI: 10.1109/ICRAIE.2014.6909173. arXiv: arXiv:1011.1669v3 (cit. on p. 13).

[Gil16] A. Gilchrist. *Industry 4.0: the industrial internet of things*. Apress, 2016. ISBN: 9781484220467. DOI: https://doi.org/10.1007/978-1-4842-2047-4 (cit. on p. 1).

[GLM08] P. Godefroid, M. Y. Levin, and D. a. Molnar. "Automated Whitebox Fuzz Testing." In: *Network and Distributed System Security Symposium (NDSS)* 9.July (2008), pdf. ISSN: 1064-3745. DOI: 10.1007/978-3-642-02652-2_1. (Cit. on pp. 13, 71).

[Gos+18] J. Gosling et al. *The Java Language Specification*. 2018. URL: https://docs.oracle.com/javase/specs/jls/se8/html/index.html (visited on 08/20/2018) (cit. on p. 19).

[GP15] K. Goseva-Popstojanova and A. Perhinschi. "On the capability of static code analysis to detect security vulnerabilities." In: *Information and Software Technology* 68 (2015), pp. 18–33. ISSN: 09505849. DOI: 10.1016/j.infsof.2015.08.002. (Cit. on pp. 2, 11).

[GS17] N. Grech and Y. Smaragdakis. "P/Taint: Unified Points-to and Taint Analysis." In: *Proc. ACM Program. Lang.* 1.OOPSLA (Oct. 2017), 102:1–102:28. ISSN: 2475-1421. DOI: 10.1145/3133926. (Cit. on p. 69).

[HCF05] V. Haldar, D. Chandra, and M. Franz. "Practical, dynamic information flow for virtual machines." In: *2nd International Workshop on Programming Language Interference and Dependence* (2005). (Cit. on p. 72).

[HM18] T. Hartmann and Z. Majó. *The Java HotSpot VM Under the Hood*. 2018. URL: https://www.ethz.ch/content/dam/ethz/special-interest/infk/inst-cs/lst-dam/documents/Education/Classes/Spring2018/210_Compiler_Design/Slides/2018-Compiler-Design-Guest-Talk.pdf (cit. on p. 25).

[HN17] F. Holik and S. Neradova. "Vulnerabilities of modern web applications." In: *2017 40th International Convention on Information and Communication Technology, Electronics and Microelectronics, MIPRO 2017 - Proceedings* (2017), pp. 1256–1261. DOI: 10.23919/MIPRO.2017.7973616 (cit. on p. 72).

[HOM06] W. G. J. Halfond, A. Orso, and P. Manolios. "Using Positive Tainting and Syntax-aware Evaluation to Counter SQL Injection Attacks." In: *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. SIGSOFT

'06/FSE-14. Portland, Oregon, USA: ACM, 2006, pp. 175–185. ISBN: 1-59593-468-5. DOI: 10.1145/1181775.1181797. (Cit. on p. 73).

[HVO08] W. G. J. Halfond, J. Viegas, and A. Orso. "A Classification of SQL Injection Attacks and Countermeasures." In: *Preventing Sql Code Injection By Combining Static and Runtime Analysis* (2008), p. 53. DOI: doi=10.1.1.95.2968 (cit. on p. 10).

[HW08] P. Hope and B. Walther. *Web security testing cookbook: Systematic techniques to find problems fast*. " O'Reilly Media, Inc.", 2008 (cit. on p. 12).

[IYJ17] J. Im, J. Yoon, and M. Jin. "Interaction Platform for Improving Detection Capability of Dynamic Application Security Testing." In: *Proceedings of the 14th International Joint Conference on e-Business and Telecommunications (ICETE 2017) - Volume 4: SECRYPT, Madrid, Spain, July 24-26, 2017*. 2017, pp. 474–479. DOI: 10.5220/0006437104740479. (Cit. on p. 14).

[Kho+11] N. Khoury et al. "An analysis of black-box web application security scanners against stored SQL injection." In: *Proceedings - 2011 IEEE International Conference on Privacy, Security, Risk and Trust and IEEE International Conference on Social Computing, PASSAT/SocialCom 2011* (2011), pp. 1095–1101. DOI: 10.1109/PASSAT/SocialCom.2011.199 (cit. on p. 71).

[LB15] T. Lindholm and A. Buckley. *The Java Virtual Machine Specification Java SE 8 Edition*. 2015. URL: https://docs.oracle.com/javase/specs/jls/se8/html/index.html (cit. on pp. 16, 19, 20, 23, 24, 26, 57).

[LL05] V. B. Livshits and M. 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. Baltimore, MD: USENIX Association, 2005, pp. 18–18. (Cit. on pp. 10, 69).

[LLW14] Liu Qiang, Liu Li, and Wang Chunlei. "Automatic fuzz testing of web service vulnerability." In: *2014 International Conference on Information and Communications Technologies (ICT 2014)* (2014),

pp. 1.035–1.035. DOI: 10.1049/cp.2014.0589. (Cit. on pp. 13, 71).

[MIT19] MITRE. *CVE*. 2019. URL: https://cve.mitre.org/ (cit. on p. 8).

[MNV14] M. Monshizadeh, P. Naldurg, and V. N. Venkatakrishnan. "MACE: Detecting privilege escalation vulnerabilities in web applications." In: *Proceedings of the ACM Conference on Computer and Communications Security* (2014), pp. 690–701. ISSN: 15437221. DOI: 10.1145/2660267.2660337. (Cit. on p. 70).

[MPL04] W. Masri, A. Podgurski, and D. Leon. "Detecting and debugging insecure information flows." In: *15th International Symposium on Software Reliability Engineering* May (2004). ISSN: 1071-9458. DOI: 10.1109/ISSRE.2004.17 (cit. on p. 72).

[NJ14] J. P. Near and D. Jackson. "Derailer: Interactive Security Analysis for Web Applications." In: *ASE '14 Proceedings of the 29th ACM/IEEE international conference on Automated software engineering* (2014), pp. 587–597. DOI: 10.1145/2642937.2643012. (Cit. on p. 70).

[NS05] J. Newsome and D. X. Song. "Dynamic Taint Analysis for Automatic Detection, Analysis, and SignatureGeneration of Exploits on Commodity Software." In: *NDSS*. Vol. 5. Citeseer. 2005, pp. 3–4 (cit. on p. 72).

[Oraa] Oracle Inc. *HotSpot Architecture*. URL: https://www.oracle.com/technetwork/java/whitepaper-135217.html (visited on 08/30/2018) (cit. on pp. 16, 24).

[Orab] Oracle Inc. *Jar File Specification*. URL: https://docs.oracle.com/javase/8/docs/technotes/guides/jar/jar.html (visited on 08/30/2018) (cit. on p. 17).

[PZK16] M. Parvez, P. Zavarsky, and N. Khoury. "Analysis of effectiveness of black-box web application scanners in detection of stored SQL injection and stored XSS vulnerabilities." In: *2015 10th International Conference for Internet Technology and Secured Transactions, ICITST 2015* (2016), pp. 186–191. DOI: 10.1109/ICITST.2015.7412085 (cit. on p. 71).

[TCJ14] M.-T. Trinh, D.-H. Chu, and J. Jaffar. "S3: A Symbolic String Solver for Vulnerability Detection in Web Applications." In: *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security - CCS '14* (2014), pp. 1232–1243. ISSN: 15437221. DOI: 10.1145/2660267.2660372. (Cit. on pp. 11, 69, 70).

[Tho+18] J. Thomé et al. "Security slicing for auditing common injection vulnerabilities." In: *Journal of Systems and Software* 137 (2018), pp. 766–783. ISSN: 01641212. DOI: 10.1016/j.jss.2017.02.040 (cit. on p. 69).

[Tri+09] O. Tripp et al. "Taj: Effective Taint Analysis of Web Applications." In: *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation - PLDI '09* June (2009), p. 87. DOI: 10.1145/1542476.1542486. (Cit. on p. 70).