Karlsruher Institut für Technologie

Institut für Programmstrukturen
und Datenorganisation (IPD)

Lehrstuhl Prof. Dr.-Ing. Snelting

Juturna: Lightweight, Pluggable and Selective Taint Tracking for Java

Masterarbeit von

Florian Dominik Loch

an der Fakultät für Informatik / in Kooperation mit der SAP SE

Erstgutachter:	Prof. Dr.-Ing. Gregor Snelting
Zweitgutachter:	Prof. Dr. Bernhard Beckert
Betreuer KIT:	Dipl.-Inform. Martin Mohr
Betreuer SAP SE:	Dr. Martin Johns

Bearbeitungszeit: 12. Juli 2017 – 19. Februar 2018

Erklärung

Hiermit erkläre ich, Florian Dominik Loch, dass ich die vorliegende Masterarbeit selbstständig verfasst habe und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, die wörtlich oder inhaltlich übernommenen Stellen als solche kenntlich gemacht und die Satzung des KIT zur Sicherung guter wissenschaftlicher Praxis beachtet habe.

Ort, Datum	Unterschrift

Abstract

Injection-Angriffe, wie bspw. *SQL-Injection* oder *Cross-site Scripting*, sind vielfältig und stellen eine große Bedrohung für (Web-)Applikationen und ihre Nutzer dar – für Privatpersonen gleichermaßen wie für Unternehmen. Bisherige Arbeiten haben klar gezeigt, dass die Idee des Taint-Trackings – Werte zu markieren und ihren Fluss durch die Applikation zur Laufzeit zu überwachen – effektiv sind bzgl. der Erkennung einer Vielzahl solcher Verwundbarkeiten sowie der Verhinderung einer Ausnutzung.

Allerdings sind aktuelle Taint-Tracking-Systeme für *Java SE/Java EE* noch nicht wirklich für den produktiven Einsatz geeignet. Die vorliegende Arbeit setzt sich mit den Anforderungen eines solchen Umfeldes auseinander und stellt, in Form von *Juturna*, einen neuen Ansatz vor. Dieser versucht, jenen Anforderungen unter Verwendung etablierter Techniken und neuer Ideen gerecht zu werden.

Eines der großen, aktuellen Probleme von Taint-Tracking ist dabei der immanente Overhead bzgl. Berechnungsaufwand und Speicherverbrauch. Daher besteht Juturna nicht nur aus einem funktionsreichen, aber dennoch effizienten, dynamischen Taint-Tracking-System, welches versucht möglichst "minimal-invasiv" gegenüber einer zu beschützenden Applikation und dem JRE zu sein, sondern enthält auch optionale Techniken aus dem Bereich der statischen *Informationsflusskontrolle (IFC)*. Dieser Ansatz soll Bereiche innerhalb einer Applikation finden, welche garantiert frei von maliziösen Flüssen sind. Dadurch wird ein selektives Taint-Tracking mit reduziertem Overhead realisiert. Juturna wird somit, sofern gewünscht, zu einem hybriden System, welches sowohl dynamische als auch statische Analyseverfahren verwendet.

Injection attacks, like *SQL Injection* and *Cross-site Scripting*, are a massive threat to the (web) applications and its users – individuals, institutions and companies. They are manifold and might occur at many places in modern (web) applications. Previous works have proven taint tracking, tagging value and monitoring their flow at runtime, to be an effective measure to detect a whole bunch of such vulnerabilities and prevent their exploitation.

But systems available for *Java SE/Java EE* today are not yet ready for being used in production contexts. In the thesis at hand, this usage scenario and the according requirements will be analyzed. Then, *Juturna*, a new take on taint tracking for Java combining established concepts with novel ideas, will be drafted, implemented and evaluated in depth with real-world requirements in mind.

One of the major problems of taint tracking is its intrinsic computational overhead and the increased memory footprint. Therefore, Juturna does not only consist of a sophisticated, yet efficient taint tracking system trying to be lightweight and pluggable by neither invasively modifying applications or the JRE, but additionally includes optional techniques known from static *Information Flow Control (IFC)*. Juturna uses these to determine execution paths guaranteed to be free from malicious flows on which tracking can be selectively avoided, reducing the caused overhead.

This makes it, if desired, a hybrid approach combining both dynamic and static analysis techniques.

Vorwort & Danksagung

Die vorliegende Arbeit umfasst sowohl die Konzeption als auch Implementierung und Evaluierung eines taint tracking-Systems in Java mit dem Zusatz, dass durch Einbindung von Verfahren zur statischen Analyse ein zu erwartender Overhead reduziert werden soll. Der Prototyp stellt, zusammen mit der vorliegenden Ausarbeitung, das Ergebnis meiner Masterarbeit am Lehrstuhl für Programmierparadigmen - IPD Snelting dar.

Ich möchte mich daher an dieser Stelle ganz herzlich bei allen Mitarbeitern des Lehrstuhls, Kollegen, Verwandten und Freunden bedanken, die zu einem – hoffentlich erfolgreichen – Abschluss dieser Masterarbeit beigetragen haben.

Mein Dank gilt Martin Mohr (IPD), Dr. Martin Johns (SAP SE) und Martin Hecker (IPD) für Ihre Beiträge zur Themenfindung. Ersteren möchte ich inbesondere für die Betreuung während der Bearbeitung, die vielen guten und konstruktiven Konversationen, hilfreichen Antworten und Anregungen danken.

Ebenfalls danke ich Herrn Prof. Dr.-Ing. Snelting für die Betreuung der Arbeit, die Unterstützung während des Anmeldeprozesses und die Bereitstellung des *JOANA*-Frameworks zur Implementierung und Evaluierung eines neuartigen, selektiven Ansatzes.

Die Arbeit erfolgte in Kooperation mit einer für die Erforschung von Sicherheitskonzepten zuständigen Abteilung der SAP SE – mit der klar bekundeten Hoffnung, die zu erwartenden Ergebnisse im Rahmen der eigenen Produktentwicklung zumindest grundlegend weiterverwenden zu können.

Unter Berücksichtigung dessen wurden nicht nur den motivierenden, konzeptionellen und evaluierenden Abschnitten der schriftlichen Ausarbeitung große Aufmerksamkeit gewidmet, sondern auch der prototypischen Implementierung, den dabei aufgetretenen Problemen, getroffenen Entscheidungen und anderen praktischen Aspekten.

Florian D. Loch,
Karlsruhe, den 19. Februar 2018

Contents

1. Introduction

1.1. Motivation

> "The impact of using unvalidated input should not be underestimated. A huge number of attacks would become difficult or impossible if developers would simply validate input before using it. Unless a web application has a strong, centralized mechanism for validating all input from HTTP requests (and any other sources), vulnerabilities based on malicious input are very likely to exist." *The Open Web Application Security Project* [1]

Injection vulnerabilities are widespread in software development: They often occur in the frontends of modern web applications, then labelled as *Cross-site Scripting* (XSS), or in backend applications and database systems – e.g., when running poorly constructed, attacker controlled, *SQL* statements (*SQL Injection*). They are a massive threat, especially in the web context.

Accordingly, the *Open Web Application Security Project* (OWASP) ranks *injection attacks* again on first place in the current release of their report *OWASP Top 10 – The Ten Most Critical Web Application Security Risks* [2]. Additionally, they rank the subspecies of *Cross-site Scripting* seventh.

Modern web applications enabling the rapid realization of complex workflows in a user-friendly, cross-platform manner caught a lot of attention in the last years and they still do today. More and more traditional applications are turning from traditional desktop applications to web apps, written with the technologies of the web: *JavaScript*, *HTML* and *CSS*. And some of those are especially prone regarding such injection attacks.

The potential damage caused by them is enormous. They might be abused to impersonate as someone else and to subsequently operate in his authentication context, or as a starting point for taking over whole systems. As those incidents almost always – either directly or indirectly – result in a financial loss for the software vendor or operator, motivation to resolve these issues can be expected.

The good news is that most of these flaws can be easily fixed after being detected or disclosed. But the bad news is that they are very easy to introduce. A lot of developers, especially those with no background in web development or interpreted scripting languages in general, tend to be not (fully) aware of this menace and therefore may not pay enough attention in order to prevent them in the first place.

Furthermore, even experienced and skilled developers are not beyond missing a vulnerability when it comes to more complex situations.

Therefore, it is important to provide developers with the right training and learning materials on the one hand, but also with mature libraries, frameworks and tools for detecting possible weaknesses on the other. For some manifestations of injection attacks, there is a "silver bullet": e.g., *prepared statements* are a great mechanism to prevent SQL injections. But for others, e.g., the various kinds of XSS, there is no such general "magic cure".

Taint tracking has been proven, under laboratory conditions and in real-world field studies, to be an effective mechanism to detect and possibly also prevent, or at least mitigate, a wide range of injection attacks including popular ones like forms of XSS [3, 4, 5]. But nowadays taint tracking approaches (for *Java*) have some downsides as explained in the next section. The goal of this thesis will be trying to resolve these and to propose a system designed for production use-cases.

Returning to the opening quote, the results of this work cannot make developers validate input cautiously – but it might be of some help finding some flaws inadvertently introduced and might prevent their exploitation.

1.2. Problem statement & research question

Although there has already been done a lot of research regarding taint tracking on different levels – ranging from special hardware, over modified runtime environments to automated instrumentation of an applications compilation – most of today's implementations still have some serious drawbacks, possibly preventing their usage in a production use-case:

- **Overhead:** Tracking of the taint information has to be done as part of the applications execution at runtime. Therefore, adding a computational overhead by additional instructions performing the propagation and an increased memory footprint for actual storing of the information.
- **Deployment:** Another major downside is that many approaches adding a taint tracking mechanism to an existing runtime environment like *Java* are very invasive by changing the runtime in a fundamental, not "pluggable" way. By this, they harm portability and especially maintainability due to closely coupling the taint tracking mechanism with the original runtime, which makes applying official updates and patches tedious and error prone.

The overhead is something intrinsic to any dynamic taint tracking system – and the more information flows are tracked, the more overhead is caused. Most of the implementations available today, see section 2.5, keep track of each and every data

flow inside an application in order to make them sound, safe and desirable from a security point of view[1].

Therefore, this thesis' objective is, beside primarily prototyping a solid tracking system, to extend it in a way that only the information flows possibly leading to injection attacks are tracked. The associated research question is how such a taint tracking system, fulfilling requirements of real-world use-cases defined in the further course, can be implemented and whether its overhead is acceptable in the given context. The second major question is, whether this overhead might be reduced by combining the dynamic *Information Flow Control* technique of taint tracking with static *Information Flow Control* forming a hybrid system while, unlike former research, putting the focus on the taint tracking system[2].

Differing from all former research done in the area of taint tracking with Java, this work will consider ease of deployment, being lightweight, compatibility and "pluggability" with decent *Java Runtime Environment*s (JREs) and Java EE application servers while being efficient and precise as important characteristics of *Juturna*. This shall enable the realization of a prototypical tool that actually can be given to developers and administrators as a tool to fight injection vulnerabilities and that can be deployed in real-world scenarios.

Taint tracking might be used to fight various kind of injection attacks, not only its sub-category of XSS attacks. Still, this work will focus on them and especially the context they appear in – but the basic system itself shall still be able to be applied to other manifestations with little or even no additions to its codebase.

1.3. Structure of the thesis

This document consists of six parts: introduction, background, concept, implementation, evaluation and conclusion. The order tries to reflect the chronology of the single phases during elaboration.

The background chapter will introduce the terms and (theoretical) concepts needed in order to follow the ideas and solutions described in the later chapters. It also covers a comprehensive summary of the so far research in the area of taint tracking and points out where this thesis distinguishes itself.

"Concept" can be considered the core part of the thesis. It will cover the attacker model, usage scenario and will, based on these, stepwise develop the design and architecture of Juturna before transitioning to sections explaining the concepts and additional ideas included.

After laying the foundations, the "Implementation" chapter will dive a little more

[1] As we will see later, this does not necessarily mean that they track all kind of data (types) available in a Java application.

[2] The reasons for focussing on taint tracking are some known limitations of static approaches regarding patterns and functionality used in modern web applications and frameworks. We will come back to this several times, but the basic decision will be discussed in section 3.2

into details of the source code written for this thesis. Some topics needing discussion on both the conceptional and and implementation level are therefore mentioned in both. Concepts and ideas that had to be very abstract before in order to forestall, now get defined more precisely.

Once the prototype's mechanisms are explained and essential implementation details have been shown, Juturna will be evaluated regarding its actual capabilities on detecting real-world vulnerabilities in some small examples in "Evaluation". Furthermore, the performance will be benchmarked in order to answer the questions formulated before: Does the approach offer an acceptable overhead while fulfilling the other requirements? Can the taint tracking be extended with static mechanisms and does this reduce the overhead? Following the evaluation, open issues and possible solutions will be discussed.

Finally, the work presented will be summarized and critically discussed before ending with some ideas on future enhancements and an estimate regarding the future of detecting injection vulnerabilities in "Conclusion".

1.4. Terms & conventions

Throughout this document the terms "string" and "string-like type" will be used in their broader sense of describing a sequence of characters or a a data structure capable of maintaining one. In the context of Java it therefore is not a short form of `java.lang.String`, it rather describes the group of `java.lang.String`, `java.lang.StringBuilder` and `java.lang.StringBuffer`. In case of non-obvious or ambiguous usage of references, the *Fully Qualified Name* (FQN) will be used.

When referring to methods, parameters are omitted unless they are needed for distinguishability or understandability.

Some more terms like "tainted", "taint-aware", augmentation, etc., and their meaning in the context of this thesis will be introduced in later sections as they need some previous explanations. The term "taint tracking" will always refer to the dynamic analysis. "Static analysis" will be used to refer to the static information flow analysis performed by *JOANA* introduced later.

When talking about "web applications" in the following, this solely describes server-side applications running in a Java EE context.

The system's name, "Juturna", is based on the same-named Roman goddess "of fountains, wells and springs" [6]. Coupled with the perception that the name of a proper Java software has to start with "J", it seemed to fit – at least to the author.

2. Background

2.1. Java EE

Beside being a programming language, Java is also a programming platform existing in different flavors in order to suit different areas of application. All of these platforms contain a set of APIs offering (basic) functionality and a *Java Virtual Machine* (JVM) capable of executing *Java bytecode*.

Java EE (Java Platform, Enterprise Edition) is build on top of *Java SE* (Java Platform, Standard Edition), containing the same JVM but offering significantly more APIs [7]. These offer advanced functionality to ease developing "distributed, transactional, and portable applications that leverage the speed, security, and reliability of server-side technology" as *Oracle* puts it [8].

Java EE is maintained by Oracle, but the project is controlled by the *Java Community Process* in order to reflect the industry's needs. In 2017, the decision was made to hand the whole project over to the Eclipse Foundation [9].

The APIs coming along with an implementation of the Java EE standard provide functionality like handling of *HTTP(S)* request, *WebSocket* communication, *JSON* processing, data persistence and much more. In the context of this work, the *Java Servlet Specification* in version 4 [10] is of special interest as it defines the (low-level) interface for handling HTTP(S) communication.

Different than with Java SE, many of the APIs added by Java EE are just specifications defining an interface but not providing an implementation. This has to be provided by the respective implementation of the Java EE standard. There are several products being compliant to all of the specifications. Examples are *Glassfish*, Oracle's open source reference implementation and *JBoss Enterprise Application Platform* from *Red Hat*.

In addition to the "full" Java EE compliancy, there is the *Java EE Web Profile*, a subset of the first in terms of APIs provided/required. These implementations are usually called *servlet containers*, although *web container* seems to be the official term [11].

Their task is to provide an environment for applications developed using the given set of APIs and to handle their execution. One of those is *Apache Tomcat*, which will be used for evaluating the presented taint tracking system in a web scenario.

2.2. Injection attacks

2.2.1. The fundamental principle

Injection attacks are a class of extremely popular and widespread attacks towards an applications by influencing its execution in a malicious way, not envisaged by the developer.

Their relevance is highlighted by various experts and organizations, e.g., the OWASP ranks them first in their *OWASP Top 10 2017* report.

The *Common Weakness Enumeration* (CWE) lists them as *"CWE-74: Improper Neutralization of Special Elements in Output Used by a Downstream Component ('Injection')"* [12].

They are not restricted to any specific programming language, framework or technology – every system causing a downstream component to execute commands influenceable by untrusted, i.e. given by a user, input might be vulnerable. Such a downstream component could be an integrated interpreter such as JavaScript's `eval()`, which interprets a string at runtime and executes it in the script's context[1]. Further examples are external databases processing SQL queries or simply a client's browser parsing a HTML page dynamically built by a server-side application containing some user input.

Whenever an attacker is able to harm the structural integrity of a command, query, etc., by "injecting" content into a later on evaluated command, he can control the applications behavior and therefore cause enormous damage: depending on the attack's context, this might go as far as arbitrary code execution. In the web context, it usually lays the foundation for further attacks building upon this position (e.g., *Cross-Site-Request-Forgery* (CSRF)[2]).

The fundamental root cause for this weakness is "improper input validation", as already described by the title of the according CWE entry (CWE-20) [13]. Input from untrusted, potentially attacker controlled sources has to be validated and masked carefully in order to make sure that it does not influence the system's further execution in an unexpected way.

In general, the basic problem is a missing separation between the "control alphabet" and the "data alphabet". This problem can often be resolved by escaping the complete input, or, in cases where this is to restrictive, by using an intermediate language that gets transformed into a safe subset of the target language like with the simplified markup language *Markdown*[3] often getting converted into HTML – but this is not always feasible. The concept of Prepared Statements is another approach following the abstract idea of separating structure and data avoiding misinterpreta-

[1] This specific form got its own CWE, CWE-95 ("Improper Neutralization of Directives in Dynamically Evaluated Code ('Eval Injection')").

[2] CSRF is used to forge requests in the users authentification context regarding a specific origin. The forging might happen, e.g., via a faked image address.

[3] `https://daringfireball.net/projects/markdown/`

tions at the interpreter.

But often proper validation is complex and developers tend to oversee edge cases and underestimate the creativity, motivation and stamina of an attacker. For some of the various manifestations of injection attacks there is a (conceptional) "silver bullet" – like the generally accepted Prepared Statements for SQL. But for most of them, there are none. As mentioned already in the introduction of this work, taint tracking has been shown to be an effective measure for detecting manifold manifestations of Injection Attacks by several authors ([3, 4, 14]). How to achieve this will be discussed in section 2.4.

There are various concrete species of injection attacks, basically it is the same idea applied to different contexts: *Command Injection* (CWE-77), SQL Injection (CWE-89), *Resource Injection* (CWE-99), etc..

2.2.2. Cross-Site-Scripting (XSS)

XSS, recorded as *CWE-79: Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')* at the CWE database [15], is a subclass of injection attacks, in which the to-be-influenced, interpreting downstream-component is a browser, respectively its HTML and JavaScript parsers/interpreters. Less abstractly spoken, an attacker tries to make an users browser execute his own commands by injecting malicious code into an otherwise benign page, abusing the users (and the browsers) trust into this page [16].

Browsers restrict access to (sensitive) information stored on the client as well as network requests according to the *Same-Origin Policy* (SOP). The SOP is a sophisticated mechanism defining how documents and scripts can interact with resources linked to other origins than the one they were retrieved from [17]. This basically means that only documents and scripts retrieved from one and the same origin are allowed to access resources linked to this origin.

As the injected code runs in the context of a given origin, an attacker could, e.g., steel a session token stored in a cookie and impersonate as a legit user or he could send further requests and directly operate in the users authentication context. He could also modify the appearance of the site in order to, e.g., make it look like the portal of the user's online bank and make him enter his credentials and than sent them home afterwards.

In 2000, one of the first XSS vulnerabilities was publicly reported [16][4]. Since then, this way of attacking has drawn security researchers' attention and today we distinguish between three different flavors of XSS, depending on how the injection is done.

[4]One of the first according to [3].

Reflected XSS

Reflected XSS, sometimes also *Non-persistent XSS* [18, 19], is the most basic variation. The idea is, to make a server-side application include malicious code into the delivered document. This often is possible, because responding systems include URL parameters, cookie values or other content controlled by the user into the returned page.

In order to initially exploit this vulnerability, an attacker needs to influence the victim's HTTP request accordingly. In the basic case of a vulnerable URL parameter, he could simply spread a prepared link (via email, as part of a post in a bulletin board, etc.).

An example is shown in figure 2.1. This very simple example does not work in recent versions of *Apple Safari*, *Google Chrome* or *Mircosoft Edge* as their XSS filter mechanisms are capable of detecting it – it only works in *Mozilla Firefox*[5]. Still, there are many examples and articles available on how to bypass these filtering systems. Also, they are just capable of detecting (simple) reflected XSS attacks, not the persistent variation described hereinafter. Furthermore, this mechanisms primarily try to protect clients, they do no help finding such vulnerabilities on the server side.

[5]Tested with the most recent version available of each of these browsers on the 10th of November 2017.

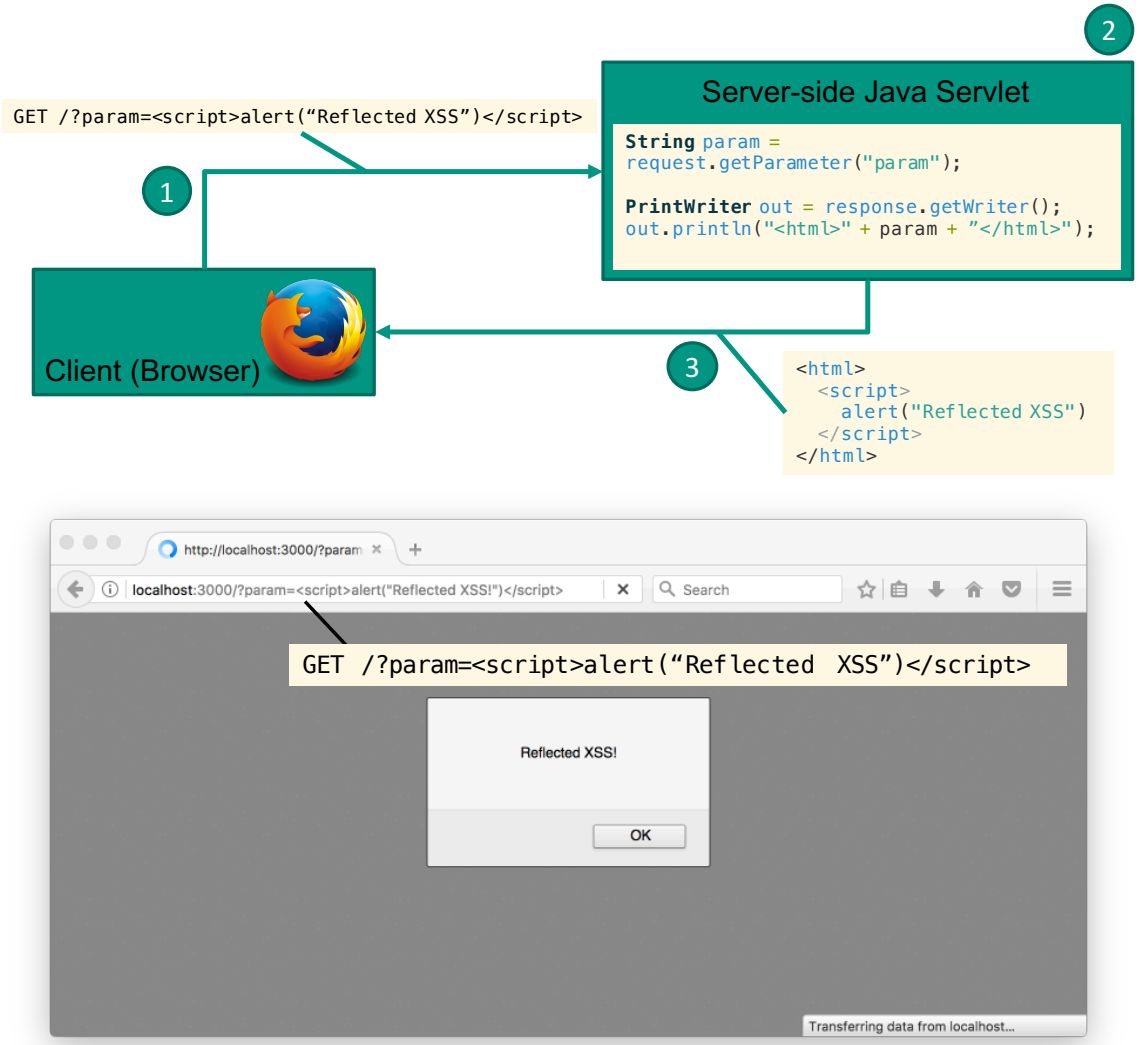

Figure 2.1.: An example of a reflected XSS attack.
Top: schematic sequence, the numbers indicate the order of the events.
Down: Exploiting this vulnerability in *Mozilla Firefox*.

Persistent XSS

At first sight, persistent XSS is the same as its reflected counterpart, but with a possibly delayed "reflection". This means, malicious content to be injected does not need to be placed in the request corresponding to a response – it might be retrieved from a database or some other persistence.

At second sight, this makes the attack much more versatile, as an attacker does not any longer need to control the victim's HTTP request as he might be able to place the exploit payload himself. A popular example for such an attack is the worm "Samy", which distributed itself through the early social network *MySpace* back in 2005 [20].

This decoupling of request and response, of cause and effect makes it impossible for the aforementioned filtering systems to reliably detect such attacks.

DOM-based XSS

In case of the two variations described above, the malicious payload inevitably has to be received and forwarded by the server-side application. With *DOM-based XSS* this is different, as this way of playing does not require a page-generating server to inject the exploit. As Klein [19], who coined the term back in 2005, points out, this subspecies differs because the malicious payload is never part of the raw HTML – it only exists in the browser's runtime representation for the document: the *Document Object Model* (DOM).

A slightly modified version of the example contained in Klein's paper (listing 2.1) demonstrates the idea and additionally shows, bearing in mind that a browser does not send the fragment portion[6] of an URL when performing a request that the server is not even receiving the injected content.

```
1  <!-- An URL might be: www.example.com/dom_based_xss.html#name=Mallory -->
2  <HTML>
3    <TITLE>Welcome!</TITLE>
4    Hi
5    <SCRIPT>
6      var pos = window.location.hash.indexOf("name=") + 5;
7      document.write(window.location.hash.substring(pos));
8    </SCRIPT>
9    <BR>
10   Welcome to our system!
11   ...
12 </HTML>
```

Listing 2.1.: An example for a DOM-based XSS attack, adjusted version of an example shown in [19].

As these attacks happen completely on the client side, their detection and mitigation is obviously not in the scope of the taint tracking system described in this

[6] Starting at the first hash/pound ("#").

work. Still, the concept of taint tracking can and has been applied on the client-side too, as shown, e.g., by Johns et al. [3].

2.3. Information flow control (IFC)

Information flow control describes the class of mechanisms able to track how information flows through an application. Research on this topic goes back to the 70's and proposed systems have evolved since than [21].

Their purpose is to analyze a given application in order to answer the question whether it handles information securely according to a given information flow policy. The two traditional policies are *confidentiality* and the dual property *integrity*. The first one assures that an application is not leaking sensitive information, the second that internal operations must not be influenceable from the outside [22, 21].

Checking, whether potentially attacker controlled input flows into an evaluation routine, as of interest in this work, is a check regarding integrity.

There are two fundamentally different variants of IFC: static/language-based and dynamic. Still, they all follow the same basic idea of annotating data with (security) labels and determining their propagation through the application. The static approaches can be subdivided again, most noteworthy exponents are systems based on dependence graphs and traditional *security-type systems*.

Systems using dependence graphs are based on the idea of modeling an application as a set of nodes connected via edges representing data dependences and control dependences. To these nodes labels can be attached, indicating a security level in order to subsequently compute the propagation of these labels in the graph. For this they utilize *program analysis* techniques [22].

Dynamic systems attach metadata to values and monitor them at runtime while security-type systems use extended type systems ensuring the given policy at compile time.

Subsequently, an advanced version of the dependence based approaches will be introduced, together with the dynamic approach of taint tracking. Security-type systems are not further discussed as they are not feasible in the given context. But before, the more general aspect of implicit and explicit flows will be explained as it is common to both.

The dependences in a program can lead to implicit and explicit flows, i.e., propagation of security labels. When information from a is copied to b (1) or a is used to determine which value to assign to b (2) there is an information flow between a and b. In the first case it is explicit, like caused by an assignment statement. In the second one it is implicit, like caused by a conditional having two branches assigning either c_1 or c_2 to b depending on the value contained in a [23].

2.3.1. PDG-based IFC

PDG-based IFC systems are an advanced variant of the dependence based approaches. As their name implies, they operate on *Program Dependence Graph*s (PDGs) using modern static program analysis mechanisms, e.g., known from *data-flow analysis* techniques, in order to check whether a given application fulfills a given security policy.

A PDG models an application using nodes for representing the program's statements or expressions, and edges for both data and control dependences [22]. PDGs have been introduced by Ferrante et al. [24]. This representation might be created from an application's source code, its compilation or, alternatively, as an intermediate program representation during compilation.

Determining how data and attached security labels are propagating through the application, respectively its PDG representation, is done using advanced *slicing* techniques.

According to Hammer and Snelting [22], total precision, i.e., no false-positives found, cannot be achieved while maintaining *correctness*, i.e, no true-positives missed, due to *conservative approximation* resulting from decidability problems.

PDG-based IFC is able to detect explicit and implicit flows in an application and can therefore be used to analyze a given program regarding the policy of *noninterference* in the context of confidentiality or integrity. It is given when there is no path between, e.g., nodes labeled as "sensitive" and ones labeled "non-sensitive". In other words, information classified sensitive does neither explicitly, nor implicitly influence variables classified non-sensitive.

As further stated by Hammer and Snelting, PDG-based systems seem to offer better precision than the security type systems by possibly being *flow-sensitive*, *context-sensitive* and *object-sensitive* resulting in a less conservative approximation.

Modern PDG-based systems often actually operate on an enhanced variant, the SDG. It gets introduced in the next subsection.

2.3.2. JOANA

JOANA "Java Object-sensitive ANAlysis" is a tool for IFC checking on applications implemented in full Java. JOANA is being developed at the chair of Prof. Dr.-Ing. Snelting for several years now and it has become a comprehensive framework for IFC-related, PDG-based static analysis. It uses the *WALA Analysis Framework* initially developed by *IBM* as a frontend for processing Java bytecode [25].

JOANA might be used in two different ways: as *Eclipse*-plugin, providing a GUI, combined with annotated source code in order to interactively perform an IFC analysis with an arbitrary security lattice, or as library providing advanced functionality and sophisticated algorithms for implementing custom analysis mechanisms. In this work only the latter is of interest.

JOANA is capable of processing full Java with unlimited threads within codebases up to 100kLOC, guaranteeing to find all explicit, implicit, possibilistic and

probabilistic flows violating the security policy. Such a flow might either violate confidentiality or, the dual property, integrity. JOANA can be used to detect both and comes with a machine-checked soundness proof [25].

As JOANA operates on Java, a fully-blown and object-oriented programming language, it needs to handle the problems arising through those: *object-* and *field-sensitivy*, *exceptions*, *dynamic dispatch* and *objects as parameters*, all leading to statically undecidable problems resulting in conservative approximation. In order to reduce the decrease in precision caused by such approximations, JOANA uses, among others, advanced *points-to* analysis algorithms in order to compute the set of possible objects a reference might actually point to [26].

JOANA works on a graph like data structure called the *System Dependence Graph* (SDG). This SDG is similar to a PDG, but beside containing the main program it additionally contains PDGs of all further procedures[7]. It therefor consists of a collection of PDGs modeling single procedures in order to allow inter-procedural analysis to be performed – usually offering massively improved results compared with earlier approaches [26].

SDGs are not a new concept, they have been brought up by Horwitz et al. [28] in 1990 already, in order to be able to compute inter-procedural slices – a major building block for most static analysis. A SDG is, as the PDG, a directed graph containing nodes and edges. The former represent a program's statements and predicates, the latter the various kinds of dependencies between them [26, 28].

When describing the selective taint tracking approach based on JOANA, some implementations details will be introduced. More information on the theoretical foundations and algorithms implemented in JOANA can be found in the publications referenced above.

2.3.3. Taint tracking

Dynamic taint tracking, sometimes also *taint checking* or *taint analysis*[8], is a purely dynamic program analysis technique used to determine the flow of tagged information through an application.

The tracking functionality is usually either embedded into the runtime environment, in case of interpreted languages, or the application itself. As it will be shown later, the tracking mechanisms capabilities vary depending on the level it is embedded into.

The entry points of a system regarding untrusted, i.e., possibly attacker controlled, input are called *sources*. Data emitted by them is *tainted*. The data is tagged[9] with *taint information*, representing its *taint state*. But as the abstract idea behind taint tracking is to attach metadata to data and propagate it along during execution, arbitrary information could, theoretically, be attached.

[7] 27, Section 14.3.1.5.

[8] In papers written by other researchers even more terms for this same concept are used, e.g., "dynamic data flow analysis" [29] or "dynamic information flow tracking" [30].

[9] Differing from static IFC, "labeled" seems to be a less common term here.

By default, a taint tracking system usually assumes data to be *untainted* unless otherwise stated. But there are also systems, like the one presented by Halfond et al. [14], using a "positive" tainting policy, i.e., tainting trustworthy content instead of untrustworthy. This will be discussed in section 3.5.1.

As the goal of taint tracking normally is to analyze whether an application's execution can be influenced by untrusted input in a malicious way, the sensitive areas of the program need to be marked. These are language, environment and application specific, but in general include functionality like runtime evaluation of strings. Just think of *JavaScript*'s `eval()` and database drivers receiving plain *SQL* commands. Such security-sensitive areas are called *sinks*. In case data, originating from a source, reaches a sink, than this is called a *(taint) flow* violating the given policy.

As the taint tracking system, as a dynamic technique, has no information regarding conditionals or other branching points, it cannot perceive the aforementioned implicit flows resulting, among others, from control dependencies. Though, it can be realized by adding static code analysis like done by Bell and Kaiser [31].

Still, taint tracking cannot, in practical terms, be used to determine noninterference regarding confidentiality or integrity as it only has information about the currently executed path and proving noninterference would require looking at all execution paths. Therefore, it can only classify a system as not complying to noninterference by detecting, at least, one taint flow.

All assurances given by taint tracking systems are only valid for the run configuration, i.e., the actual executed path. Therefore, they are not holistic regarding an application – in contrast to the ones provided by static IFC mechanisms. Taint tracking only analyzes executed code and therefore cannot find vulnerabilities off the executed path. This results in true-positives not being found (correctness).

But as taint tracking "knows" the path actually being executed, it does not need to do a conservative approximation for, e.g., all possible values $a_0, \ldots, a_n$ that could flow into a variable b due to multiple data dependences. A (basic) static mechanism needs to do so as it does not know which path out of a set of possible paths will get executed during runtime. Therefore, taint tracking achieves a higher precision than static IFC. At least in theory, taint tracking does not lead to false-positives and therefore even achieves total precision[10] [32].

Tracking only explicit flows resulting from data dependencies is considered to be sufficient for checking an application – to be more precise, the actually executed path – regarding its integrity, but subpar when checking for confidentiality as leaking "hints" about the value via implicit flows might be much more severe in this case.

As taint tracking happens at runtime, it causes a computational overhead. Additionally, storing the taint information results in additional memory being allocated.

There are different ways how to implement taint tracking systems. They will be

[10] Due to another reason also taint tracking mechanisms might need to do a conservative approximation too: when a system does not track at the highest possible granularity. This will be elaborated in section 3.5.

introduced, discussed and compared in depth in the following.

For a more theoretical introduction to taint tracking, please be referred to the comprehensive overview written by Schwartz et al. [32].

2.4. Detecting injection attacks with taint tracking

After the mechanics of injection attacks and taint tracking have been introduced in the previous sections, we can now step ahead and shortly examine how the former can be detected using the latter.

The basic idea is really simple and matches IFC's idea of ensuring integrity (for the actual executed path only): possibly attacker controlled input is tagged as tainted and therefore marked as untrusted. It is then tracked by the system while flowing through the application, having it propagate the taint information. In case the tainted value reaches a sensitive area/sink (like `eval()`), the taint tracking system can act accordingly. To indicate that an attacker controlled value has been validated and adjudged benign, the attached taint might be removed.

In the context of this work, whose purpose is to sketch and prototype a widely applicable solution for Java while using the problem of injection attacks in web applications as a motivation, entry points defined in, among others, the Java Servlet Specification [10], need to be considered (taint) sources. As client and server speak HTTP, the entry points are, among others, contained in `javax.servlet.http.HttpServletRequest`. An example for a method needing to be declared as source is `getQueryString()`.

Exemplarily targeting reflected XSS vulnerabilities, sinks inside the server-side application itself are excluded. The sinks of interest are the places where information is returned to the client. Exit points, at which tainted information might leave the web application on the way back to the client, are located in `javax.servlet.http.HttpServletResponse`[11].

With this, the following basic policy can be configured and should be enforceable by Juturna: information given in the request of a client should not be contained in the response unless validated/sanitized. We will come back to this in more detail during the evaluation of a web application in section 5.1.

Finally, to point out the usage scenario of taint tracking regarding injection weaknesses once more: Taint tracking does not make the actual codebase immune against injection attacks, like a sound sanitizing mechanism – if possible in a given context – would do. Neither does it detect these vulnerabilities in a given codebase, like a static analysis might be capable of. Instead, it provides a hardened runtime environment, which is able to recognize, stop and report malicious data flows in code actually being executed. There is no need for (manual) modifications to existing applications, so this is especially useful in case of already existing legacy applications or scenarios, where a simple "escape-everything-sanitizing" strategy is too restrictive.

[11] That is only half the story, but for keeping the example scenario concise, it has been simplified

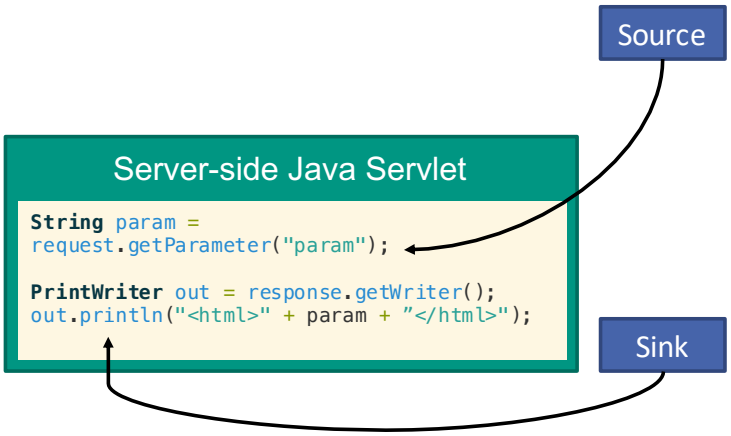

Figure 2.2.: Example of source and sink in a simplified servlet using APIs provided by the Java Servlet Specification.

2.5. Related Work

Subsequently, related research in the domain of detecting injection attacks with taint tracking and static IFC will be presented. The emphasis is put on Java SE/Java EE related solutions, other platforms and languages are out of scope. As there has been quite some research, only the more important works and the ones having similarities with Juturna are mentioned.

Summarizing the state of the art regarding research on static IFC would also go beyond the scope of this work. The same applies to listing alternative techniques for detecting injection vulnerabilities. Most of them tend to be very specific, e.g., statically building a model of valid SQL queries and checking actual queries against it at runtime as done by Halfond and Orso [5], or very abstract and therefore often imprecise, like Sekar's language independent approach intercepting (web) applications on network or library level in order to find suspicious data flows [33].

Beside the research projects mentioned in the following, there are also commercial products available often marketed under the term of *Runtime Application Self-Protection* (RASP) – e.g. the runtime protection offered by *Veracode*[12] – promoting high detection rates. But as they do not state technical details on how this is achieved, they are hard to classify. Another product, *IBM AppScan Source*[13], internally uses the later mentioned tool *Andromeda*.

Since the programming language *Perl*, respectively its interpreter, received a feature for dynamically tracking taint information along with actual values back in 1989 [34], quite some research and implementation effort has been done to provide this functionality also for applications written in other languages and running on

[12] `https://www.veracode.com/products/runtime-protection-rasp`
[13] `https://www.ibm.com/de-en/marketplace/ibm-appscan-source`

different platforms. Beside language specific approaches, there are also more holistic research projects using instrumentation on compiled binaries (e.g., [35]).

Several works brought the idea of taint tracking to the Java-world and especially to the closely linked ecosystem of *Android* (e.g., Enck et al. [36] and Weichselbaum et al. [37]) – usually focussing on the detection of sensible information leaked by apps checking for its confidentiality property in terms of IFC.

2.5.1. Purely dynamic approaches

With Java, there are two possible ways on how to implement taint tracking mechanisms, which all related projects are based on:

- **Bytecode instrumentation:** The compiled application and system classes' bytecode is modified by adding additional instructions in order to introduce, propagate and check taint information. This is a powerful and universal approach needing classfiles[14] to be editable[15]. It might be performed during an additional preprocessing step or while loading a classfile into the JVM. This mechanism is also popular with profilers or other low-level JVM monitoring tools and can be considered to be more difficult to implement.
- **Augmentation of the standard library:** Instead of modifying the compiled bytecode, the standard library classes get modified on the source level. This results in a more lightweight, less invasive and context-sensitive adjustment without needing a preceding step. Therefore, it is also less powerful: it is not possible to add tracking of taint information for primitive data types.

Beside these two and their possible combinations, there are also special research JVMs (like [38, 39]) not suitable for an in-production usage scenario. We will come back to specific advantages and disadvantages of instrumentation and augmentation later throughout the work.

Four research projects are similar to the approach presented herein with regard to the taint tracking mechanism. They have also served as sources of inspiration for Juturna. Nevertheless, comparing the respective results in more detail it becomes obvious that they are very different.

Haldar et. al. introduced taint tracking to Java in their work *Dynamic Taint Propagation for Java* [4]. They only perform string-level based tainting – which is much to imprecise for detection of injection vulnerabilities without causing too much false-positives due to conservative approximation. Therefore, as elaborated in section 3.3.2, string granularity is not considered suitable. Furthermore, they just

[14] A classfile is the artifact generated by a Java compiler. It produces one classfile for every class, interface, etc., containing the JVM bytecode and related data. Multiple classfiles usually get bundled in a *Jar* together with some descriptors.

[15] This might be more of a legal problem than a technical one.

use a `boolean` as flag for representing the taint state of the string, not enabling any differentiation and therefore no context-sensitive sanitization and mitigation.

The way their "untainting"-operation has been implemented is akin to Perl's approach: they assume every call to *regular expression* matching functionality and methods like *indexOf()* to be part of a sanitization mechanism and therefore reset the taint flag of this string. This seems to be a very questionable heuristic, as the assumption that every call of such a method is meaningful and intends to escape a string surely does not hold true for the majority of applications. As a robust alternative, a more sophisticated, flexible and configurable strategy with the goal to avoid such false-negatives will be described in this work.

Haldar et. al. equip the JRE with their mechanism by a one-time instrumentation of the system classes' bytecode. Furthermore, they do an on-the-fly instrumentation of the application's bytecode when the classfile gets loaded into the JVM providing methods marked as source or sink with additional code.

Chin and Wagner [40] present the first character-level implementation, representing the taint state of every character in a string with a binary flag in an array. Therefore, this approach is not capable of differentiating between multiple sources of taint either. Similar to Haldar et. al., they also replace some Java bytecode files, which are part of the standard library of the JRE. According to them this invasive approach did not work for all of the major JRE implementations at that time, including the one of *Sun* (today Oracle).

Another approach on making Java taint-aware comes from Bell et. al. [31] with their system *Phosphor*. Beside string-only taint tracking on a character-level, Phosphor provides taint tracking for all primitive data types and is therefore much more holistic than earlier approaches. It is also able to track implicit flows. This is made possible by a massive, low-level bytecode instrumentation causing a severe overhead – the authors state an overhead of 52 % on average for the faster configuration running the *DaCapo 9.12-bach*[16] benchmark.

In section 3.3.1, it will be discussed whether these tracking capabilities are actually necessary in order to find injection vulnerabilities, or whether a reasonable trade-off between detection rate and overhead caused can be made. They also modify the JRE installation. This can be considered the most advanced and promising approach regarding taint tracking in Java that has been published so far.

Finally, another implementation has been done by Dallermassl [41]. It differs from the others as it does not directly use bytecode instrumentation but rather uses *AspectJ*, a Java framework for *Aspect Oriented Programming* (AOP), in combination with a set of defined aspects augmenting the string related classes in the standard library.

This work is very similar to Juturna with regard to the way it makes the JVM

[16] `http://www.dacapobench.org/`

use the adjusted standard library classes. Both make use of the "-Xbootclasspath"-switch and the Java agent technique introduced later. Still, their concept is strictly limited to string-level, coarse-grained tainting making it differ quite a lot from Juturna's taint tracking engine. Fine-grained, character-level tainting for system classes, as done by Juturna, cannot be realized using an AspectJ-based AOP-driven implementation strategy.

2.5.2. Hybrid approaches combining static and dynamic analysis

Beside the purely dynamic approaches listed above, there are also purely static ones like the aforementioned, mature Andromeda project by Tripp et al. [42]. It has been developed especially with industry scale web applications in mind. Basically, it lazily computes a call-graph representation of an application and enhances. Following a demand-driven approach – performing more costly static analysis operations (like a points-to analysis) only when needed (in this case, a vulnerable information flowing into the heap) – it enriches the model of the to be analyzed application selectively and therefore, according to its authors, provides great scalability. It currently supports Java, JavaScript and *.NET* applications.

There are many more static approaches, but as Juturna's focus is clearly on dynamic taint tracking, scrutinizing them would go beyond the scope. As mentioned before, static and dynamic analysis techniques both have their weaknesses – which can partially be compensated by the other, therefore combining them seems natural. Dynamic analysis suffers from the need of having paths of an application to be actually executed in order to be checked and from the linked runtime overhead. Static checks often struggle with imprecision and false-positives caused due to the need of conservative (over-)approximations. Additionally, full language/platform support (e.g., multi-threading, Java Reflections, etc.) and scalability are issues – but therefore no runtime overhead is added and they offer complete coverage of an application.

For Java, very little research has been done regarding hybrid analysis approaches – at least compared to other languages like PHP or C[17].

The common way to form such a hybrid approach is to perform a static (dependence) analysis on the applications (byte)code and to determine security-sensitive parts. These are, put simply, the instructions handling data that emerges from a source and then flows into a sink – this will be defined more precisely when introducing the selective taint tracking solution included in Juturna in section 3.6.2. Then, it is possible to perform taint tracking only on these parts. It is directed by the static analysis.

Both of the hybrid approaches done for Java so far – by Mongiovì et al. [45]

[17]For example applied in order to find explicit buffer overflows (Aggarwal et al. [43]) or ones caused by `printf()` (Chang et al. [44]).

(*JADAL*, 2015) and Zhao et al. [46] (2016) – follow this idea[18].

Both systems are based on the idea that only the parts of an application being security-sensitive actually needs taint tracking to be performed on.

Zhao et al. determine these parts based on the idea of program slicing as described by Weiser [47]. They do not give much information on how exactly the static analysis is performed, making it impossible to judge about its precision and scalability. They use AOP for the instrumentation of the sensitive parts, resulting in a rudimentary taint tracking system.

JADAL, in a nutshell, assigns the bytecode instructions to nodes in a *Data Dependence Graph (DDG)*, representing possible paths from sources to sinks, so ones being security-sensitive. For these paths it applies dynamic taint tracking. The taint tracking component monitors taint information on a variable level and is quite uncommon: The instructions on these paths get a call to a checking function put in front. This function gets called with the current executing context (stackframe) and the respective node as parameters. It notes the invocation in the current context down in a central data structure, but only in case the preceding node on this path has also been invoked in the same context before or is a source. By this, the system stepwise tracks the execution path and is able to check at a sink whether a security-sensitive path (possibly via multiple edges) in the DDG has been used to get there.

As Mongiovì et al. do not provide performance benchmarks, scalability of JADAL's taint tracking mechanism is questionable. Also they target on detecting data leaks, the dual problem to checking for integrity.

Both works introduced focus on static analysis, assigning taint tracking just a supportive, second-tier role. Juturna does it the other way round. It focusses on a more sophisticated taint tracking mechanism, offering functionality to disable taint tracking selectively for parts of an application not considered security-sensitive by a preceding, also PDG-based, static analysis.

As it will be discussed extensively in section 5.4, after the evaluation of the presented prototype, especially static analysis techniques suffer by Java features like Reflection and custom *classloader*[19] hierarchies extensively used in modern web frameworks – although there are works like the static, type-based one of W. Huang et al. [48] claiming to resolve them (at least partially).

[18] Which has been implemented for C already 10 years earlier by Aggarwal et al. [43].

[19] A classloader is used by the JVM to load a classfile. It is responsible for providing the bytecode of the classfile identified by a given name. By providing a custom classloader, an application can influence which bytecode gets loaded into the JVM at runtime.

3. Concept

3.1. Setting the scenario

3.1.1. Use case

The system implemented as part of this thesis, Juturna, was drafted with a specific usage scenario in mind: mitigating injection attacks in Java EE servlets, potentially running in a production scenario. As described, the work shall focus on reflected XSS for demonstration purposes. However, covering other manifestations of injection attacks like command or resource injections should not need a considerable amount of additional implementation effort to be done, modifying Juturna configuration should be enough for basic detection.

This implies further requirements to the system: configurability and extendability. These abilities are also needed in order to adjust the system to a specific implementation of a Java EE servlet container like *Apache Tomcat* or *Jetty*.

Covering persistent XSS or other injection attacks requiring taint information to be persisted are out of scope, but a suggestion regarding an extension to handle them will be made very briefly in the outlook of this work section 6.3.

As described in section 2.2.2, reflected XSS exists due to unvalidated input – usually received via HTTP requests – that gets relayed by a server-side application including this input into its response. Regarding different injection attacks like command injection, we also assume HTTP requests to be the primary entry point – but as mentioned already, it shall be possible to configure the system in order to make it capable of using different entry points too.

Another requirement to the system is that it needs to operate on Java bytecode as it cannot be expected to have the actual Java source code at hand. This, for example, would be the case when including closed-source third party libraries or when hosting a customer's application.

Furthermore, as few as possible assumptions regarding the to be monitored application should be made. This especially involves the usage of *Java Reflection* for meta programming, which is very common in modern frameworks like the popular *Spring Framework*[1].

[1] `https://projects.spring.io/spring-framework/`

The purpose of Juturna shall be to help developers and administrators to find injection weaknesses caused by inadvertence, lack of knowledge or high complexity hiding them. It shall augment and harden the runtime environment, but it is not supposed to find vulnerabilities deliberately introduced by developers ("backdoors"), which would be a much harder – but also completely different – task.

Security flaws reported by the system need to be checked by a developer. Therefore, it is necessary to keep the amount of false-positives as low as possible – otherwise the system would probably be of no use.

As mentioned before, taint tracking is a popular mechanism for the detection and mitigation of injection attacks. There might be already similar tools – at least there are papers describing them – fulfilling some of the requirements stated so far – but as this thesis is written in corporation with an industry partner, the SAP SE, "some" is not enough and this work tries to provide a solution suiting their needs by coming up with new ideas and/or combining existing techniques.

Furthermore SAP wants to gather knowledge in the field of taint tracking for Java in order to implement their own, custom-tailored solutions while being independent from third-party tools.

3.1.2. Attacker model

In order to describe a protection mechanism and to show what it has to be capable of, it is essential to properly define the threat to be detected/mitigated and the according attacker performing it. After the attack itself has been described in depth in section 2.2, an attacker model will be defined now.

Differing from the well-known *saboteur* described by Dolev and Yao [49], often referred to as *Dolev-Yao-attacker*, this opponent is considered to be less powerful. Dolev and Yao concede their saboteur to have the following, active and passive, capabilities:

- He can obtain all messages sent through the network
- He is a legitimate user and can initiate conversations
- He is in the position to be a receiver to any other user in the network

As their model was drafted in order to analyze security of public key protocols communicating in an insecure environment on a network layer, it does not suite this scenario too well, still it is a good reference and starting point in order to emphasize the capabilities of an attacker.

For this work solely HTTP(S) or similar protocols like *HTTP/2* are of interest. We assume no man-in-the-middle attacks (like they could be performed by proxies injecting malicious code into webpages). This implies that all communication is already secured against eavesdropping, intercepting or packet injection by a mechanism on layer 5 of the *OSI model*, e.g., TLS, and our attacker, M, can only operate on the application layer.

As M is capable of sending arbitrary (HTTP) requests, he is able to verify and exploit weaknesses directly, depending on the kind of injection attack; by this he can also "store" malicious payloads in case of persistent attacks. Regarding the scenario of XSS, M is capable of proposing/offering a starting point for a request to a potential victim A (might be one out of many as such attacks are easy to be widely spread). A is expected to use a decent browser. Such a starting point could be a direct link sent via email, possibly resulting in a HTTP request executed in the security context of A's browser. Or it could lead A onto a website under the control of M, then used to trigger the actual "attack request" – whereby this intermediate step could be used in order to be able to perform not only *GET* but also *POST* requests, gaining control over the payload/body of a HTTP request.

Implied by this, one has to assume that M, in general, is able to control every part of a HTTP request as specified in RFC 7231 [50]) – also this might not be the case for (reflected) XSS scenarios, in which the set of controllable parts of a request is significantly limited. Still, this requires carefully studying the Java Servlet Specification regarding entrypoints.

Applying the principle of Kerckhoff[2] to this scenario, security of an application should not be depending on its source code not being public. Therefore, M is acknowledged to know about the input, side effects and output of a system to be attacked as well as about its internal logic. This enables him to actually find weaknesses in an application.

3.2. Why to choose taint tracking instead of static IFC?

Subsequently, the floating question of "why taint tracking and not static IFC?" will be answered by briefly pointing out some important advantages of the former, regarding the given scenario, and linked implications. This merely draws conclusions from both approaches' characteristics described before.

With taint tracking, one does not know about vulnerabilities until execution of

[2]Kerckhoff's principle states that the security of a (crypto)system should only be based on the secrets used for encryption and that all further information, e.g., the algorithm used, might be publicly known. [51]

path leading to them as detection happens dynamically, at runtime. In other words, not executed program paths have not been verified regarding their innocuousness.

But it also implies that the taint tracking system – as it needs to embed code into the actual application, or the runtime environment, in order to keep track of flows, tainting at sources and checking at sinks – is able to prevent the exploitation of a spotted vulnerability almost out-of-the-box. This possibly enables an administrator or an automated service (e.g., the very popular *Platform as a Service* (PaaS) environments) to run an application containing such weaknesses in a safe manner – in a given scenario this might be helpful. Static IFC takes all paths into account. In order to add code for mitigation of spotted weaknesses, it needs additional functionality for modifying the application.

Considering every situation in which a tainted string reaches a sink to be an actual incident (true-positive), a proper taint tracking system using character-level precision does not yield false-positives as it does not need to do a conservative approximation at any time. This results from the fact that it only has to look at one concrete execution path. It simply does not matter for the taint tracking system by how many paths a given execution point might be reached as it (only) knows about the one actually being taken, not having a holistic view on the application.

Because of this, dynamic taint tracking systems are independent from the size of an application. Precision and caused overhead do not depend on it. Static approaches using a graph-like, in-memory representation of the to-be-checked application are however limited in terms of an application's size.

But static IFC has to offer much stronger security guarantees: beside explicit flows it is also capable of detecting implicit flows and can therefore make a statement regarding non-interference between input and output. But as implicit flows are not essential for the detection of injection attacks one can get over this disadvantage of dynamic approaches[3].

The actually determining factor deciding in favor of a dynamic approach is the general support for meta programming techniques, dependency injection, complex classloader hierarchies and further mechanisms becoming more and more popular in (large) web applications, which are often transparent for taint tracking but major show stoppers for static IFC. Still, static IFC approaches are evolving and they provide great characteristics. Therefore, an extension making Juturna a hybrid approach will be suggested in section 3.6.2.

To point it out once more: static IFC and dynamic taint tracking are two fundamentally different approaches sharing some objectives. In theory, static IFC is

[3]Bell and Kaiser [31] showed that, adding a preprocessing step performing static code analysis and adjusting the instrumentation accordingly, dynamic taint tracking can also track implicit flows.

superior, in practical terms taint tracking may be in the lead. This makes it very hard to compare them and impossible to declare a superior technique as both have unique strengths and capabilities. Therefore, as we will see in section 3.6.2, combining both approaches – static IFC and dynamic taint tracking – might be the best idea towards more efficient taint tracking. Due to the practical benefits of taint tracking considering the given scenario, it will the primary mechanism of Juturna.

3.3. What to track? Designing a taint tracking system

3.3.1. Benefits and drawbacks of string-only tracking

Following from the fact that, as described in the use case above, input as well as output are both of a string-like type, the system is restricted to only track taint information for those: `java.lang.String`, `java.lang.StringBuilder` and `java.lang.StringBuffer`. This does include array types based on them.

Refraining from monitoring other data types, like the primitive ones (`int`, `char`, etc.) and their object and array types, one might lose soundness and universality. But therefore one can create a system operating with less overhead while still being suitable for the specific problem given – the detection of injection attacks in Java (web) applications.

From a security point of view this approach might sound odd, but taking into account the performance and scalability aspects required for production usage, a decreased recall can be considered acceptable. It is on dice that this will, in general, massively reduce the overhead of the taint tracking mechanism – whether this decrease is enough and the remaining overhead therefore acceptable in the end, will be discussed in the evaluation (see chapter 5).

```
1  String in = "foobar";
2
3  taint(in);
4
5  String copy = "";
6  for (int i = 0; i < in.length(); i++) {
7      copy = copy.concat(in.charAt(i)); // String.concat() receives a primitive char
8      // without taint information attached
9  }
10
11 assert(copy.equals("foobar")); // true, copied String char-wise
12 assert(isTainted(copy)); // assertion fails, copy is not tainted
```

Listing 3.1.: A Java snippet loosing taint information in a string-only taint tracking scenario. `taint(v)` is a fictive function marking the (complete) content referenced by `v` as tainted.

Listing 3.1 makes the problem of not tracking primitives more vivid: one can see that taint information gets lost during execution by copying a `java.lang.String` in a primitive, char-wise way. As, according to the proposal formulated above, only the string-like types carry along taint information, it is lost as soon as a single, primitive `char` get accessed.

But as mentioned before, the systems shall help developers and administrators to find vulnerabilities negligently introduced into an application. It expects developers to stick to best practices, and code like this would possibly violate them and should not exist in a properly engineered application. This might sound rough and of course, there are legit scenarios in which such code might exist, e.g., in libraries offering advanced, high performance string manipulation. But in such cases, this should happen at a central position and the code could be adjusted in order to lose the taint information. How this works can be read in the "Implementation" chapter. Also it has been stated before, that as few as possible assumptions should be made regarding an application to protect, this one needs to be made. In the present scenario it is assumed to have mostly high-level business logics and "low-level" functionality is expected to be provided by libraries which can be adjusted in an one-time effort.

This decision also has implications regarding the implementation of the taint tracking mechanism as it allows strategies to be used, which, including the one that has been chosen for Juturna, are not capable of tracking information for primitive types.

To conclude: Even if the system occasionally misses an actual weakness, due to this deliberate restriction, seems acceptable as part of the trade-off made. Using a system not detecting all vulnerabilities still can be considered a much better situation then using no detection mechanism at all. And a more sound approach causing massive overheads, as measured by Bell and Kaiser [31] (see section 2.5), or containing restrictions regarding used technologies, e.g., Java Reflection, might not be bearable in a production setup at all.

3.3.2. Granularity matters: string-level vs. character-level

String-only taint tracking is split in two factions regarding how precise the tainting can be done: on string-level[4] or on character-level[5]. In case of the former, the smallest entity in the taint tracking policy is a string – it is either completely tainted or not at all. This might lead to a system consuming less memory, but at the expense of the mechanism's precision.

[4]Haldar et al. [4], Dallermassl [41]
[5]Chin and Wagner [40], Bell and Kaiser [31]

```
1  String a = "foo";
2  String b = "bar";
3
4  taint(b);
5
6  String c = a.concat(b);
7
8  String d = c.substring(0, 2)
9
10 assert(d.equals("fo"));
11 assert(isTainted(c)); // true for both granularities
12 assert(isTainted(d)); // true in case of string-level and false for character-level
   tainting
```

Listing 3.2.: An example leading to a false-positive, introduced due to string-level tainting granularity.

Listing 3.2 contains a small example, that makes clear why tracking taint information on a character-level is crucial in order to create a precise system, easily avoiding a class of false-positives. In the listing a substring is created from the concatenation of two other strings, out of which one is tainted.

In this example the (binary) taint flag of a concatenated entity `s = concat(a, b)` is set by `taint(s, isTainted(a)` $\vee$ `isTainted(b))` in case of string-level granularity, performing a conservative approximation. With character-level tracking an array might be used in order to store the taint information for every `char` contained in the string. Therefore, when concatenating two strings, the taint information array would simply be concatenated too. The `substring()` method returns a tainted string in case the string called on is tainted, respectively uses a slice of the array containing the taint flags when operating on character-level. `isTainted()` now returns `true` if at least one character of the string is tainted and `false` otherwise.

We now "execute" the example assuming the two different precision levels. With string-level granularity the c becomes tainted through the concatenation – marking characters tainted which could actually be considered safe ("overtainting"). Schwartz et al. use the term "taint spreading" to describe this problem [32]. Creating the substring d results in another tainted string – although it does not contain a single character of the, potentially attacker controlled, input. If this substring, d, flows into a sink, the system would yield an (unnecessary) alert. With character-level granularity, this does not happen as only the second half of c is tainted, all characters extracted into d therefore are "clean". Overtainting does not occur, avoiding possibly plenty false-positives.

Because of this, Juturna uses a character-level tainting policy.

3.3.3. What information to be attached?

The term of "taint information" has been used so far, without actually explaining what kind of information actually gets attached. This may forestall some ideas introduced later, but influenced subsequent decisions and therefore should be briefly mentioned.

Taint tracking systems suggested for Java so far, except the work of Bell and Kaiser [31], are only capable of flagging data, i.e., they attach a binary taint state. This is a good foundation, but does not enable differentiation between different sources introducing taint into a system. But distinguishing between them lays the foundation for specific handler routines making the system not only capable of detecting injection attacks but also of mitigating them.

As we will see later, the implementation of Juturna is even capable of attaching additional, optional debug information or possibly arbitrary data in an efficient manner.

3.4. Choosing an implementation strategy

Before we are going to discuss different implementation strategies and finally present the one used by Juturna, we are going to recapitulate the requirements towards the to-be-drafted system which have been verbalized so far.

- Support for existing servlet containers (like Apache Tomcat or *Jetty*) implementing the Java EE Web profile (a subset of the Java EE specification) shall be provided at least on a conceptional level. The implementation shall not target them explicitly – support shall rather be provided by offering a decent grade of configurability and extendability.
- Char-level granularity on string-like types shall be enabled.
- Shall be portable and pluggable. Shall not need (invasive) changes to the JRE installation. Shall use, preferably, standardized mechanisms in order to be "JVM-agnostic" and compatible with "of-the-shelf" JVMs.
- No preprocessing of applications should be needed. As few as possible assumptions shall be made regarding the to-be-executed applications, therefore especially Java Reflection shall be supported as it is very popular with modern (web) frameworks.
- Good performance and reduced memory footprint are important as the system shall be used in production contexts.
- Source code of applications should be considered not available. Java bytecode has to be sufficient for the system.

- The implementation should be easy to maintain by sticking to common best practices of software engineering.
- It should be possible to store not only a binary taint flag as taint information, but data originating from different kind of taint sources should be marked differently.

Beside some fancy strategies, like encoding the taint state in the *Unicode* codepoint[6] [52] and the usage of special research JVMs not suitable here, there are two ways of bringing taint tracking to the Java world suggested so far: bytecode instrumentation and modifications/augmentations to the Java Runtime Environment, more precisely the standard library/classes shipped with it. They have been introduced in section 2.5.

In accordance with the requirements listed above, it was chosen to go with a combination of both approaches. How they have been combined in Juturna will be explained in the following while also pointing out some differences to similar approaches by other researchers which are generally discussed and introduced in the section on related work (section 2.5). To the authors best knowledge, Juturna is the first system combining both in this fashion.

3.4.1. Augmentation on source level

In order to make the string-like types (`java.lang.String`, `java.lang.StringBuilder` and `java.lang.StringBuffer`) "taint-aware", i.e., making them capable of carrying and handling taint information, the most straight-forward solution is to add another field to these classes in which the taint information is hold – regardless whether one wants to store just a single bit per character, e.g., by using a `boolean[]`[7], or a more complex structure extending `java.lang.Object`.

Juturna belongs to the latter category by using a concept named "taint ranges". For the moment it is enough to understand, that those are used to store the taint information – we will look at them in depth in section 3.6.1.

The term "augmentation" shall express, that existing Java source code gets enriched ("augmented") by hand with additions custom-tailored for a specific context. It is used in the work at hand in order to differentiate between this manual step and the automated, much more generic, process of bytecode "instrumentation".

Besides adding a field for storing the taint information, these system classes need to be augmented in order to also propagate this metadata. In other words, the

[6] Zekan et al. encode the binary information whether a character is tainted or not by shifting the Unicode codepoints of those characters. As they get shifted into an area designated for private use, the system can easily detect whether a character is tainted and shift it back in order to retrieve the actual codepoint. Because the information is encoded in the character, it can easily be persisted in databases, or exchanged with other applications. Though, the amount of information that can be represented in this way is very limited (1 bit). They implemented prototypes for Java and PHP.

[7] The usage of a `boolean[]` is not recommended. This will be explained in section 3.6.1.

methods defined on the string classes need to be adjusted so that they keep this metadata synced when modifying their content respectively their internal `char[]`.

But not only the string classes need augmentation in order to lose as few taint information during execution as possible: e.g., `java.lang.String#charAt(int position)` might be used to retrieve a single `char` which is not able to carry taint by itself. As described before, usage of these methods is expected to mostly happen in performance critical parts of the codebase, especially in libraries. The implementation of *regular expressions* contained in the JRE (`java.util.regex`) is one of those using this kind of low-level functionality internally. In order to avoid loosing taint when using Regular Expressions the according classes also needed to be augmented.

Because primary types are – differing from `java.lang.Object` and its inheritors – not defined in Java code themselves, this approach does not work for them.

The augmentation performed can be considered to add taint tracking on "library-level", positioned between bytecode-level and application-level. As we will see in the further course of this work, this offers many benefits by being able to provide special handling for at some points requiring it, while still staying on a low enough level to be transparent to most of the application code.

Making the JVM use these augmented classes

After adjusting the standard library, a proper way has to be found to make the JVM actually use these modified classes. Chin and Wagner [40] simply tried to replace the compiled classfiles inside the JRE installation – but according to them, they just found one JVM accepting this; other JVMs, including the most common ones, detected these files as modified and refused to use them.

But there is an alternative, much better way of doing this: the `java` command running an application in a newly spawned JVM offers a commandline flag – `-Xbootclasspath` – for overriding or extending the path where this single JVM instance will search for the classfiles it needs for bootstrapping. The usage of this parameter is shown in figure 3.1. This flag seems to exist since the days of Java 1.3 [53, p. 341] and is supported by at least the JVMs of Oracle, IBM and the OpenJDK project – although it is prefixed with "X", meaning it is not officially standardized.

One simply needs to compile and package the augmented classes and run a given application using this commandline option, in order to provide a basic taint-aware environment. This makes the whole approach portable, as the JRE (especially its standard classes located in `rt.jar`) do not need to be touched.

As the the additional taint tracking code is introduced on the lowest level that is possible without touching the JVM itself, one gets support for Java Reflections and other problematic features mentioned so far "for free". An example of how such an invocation of `java` does look like is shown in the next subsection in figure 3.2.

```
❯ java -X
    -Xmixed           mixed mode execution (default)
    -Xint             interpreted mode execution only
    -Xbootclasspath:<directories and zip/jar files separated by :>
                      set search path for bootstrap classes and resources
    -Xbootclasspath/a:<directories and zip/jar files separated by :>
                      append to end of bootstrap class path
    -Xbootclasspath/p:<directories and zip/jar files separated by :>
                      prepend in front of bootstrap class path
```

Figure 3.1.: Call to the `java` command printing information on the usage of the `-Xbootclasspath` parameter.

3.4.2. Bytecode instrumentation

Right now, the described system is just partially taint-aware: it is able to hold taints and propagate them along, but it has no actual sources and sinks – so basically no taint information exists that needs propagation so far.

These parts get added via bytecode instrumentation. The basic idea is quite simple: the bytecode of methods declared as sources and sinks gets processed and additional bytecode for setting a certain taint information, respectively checking for one in the case of a sink, gets inserted.

This approach got many advantages over simply adding tainting and checking functionality on the source-level (augmentation) as, e.g., done by Chin and Wagner [40]: it works with arbitrary, already existing and compiled bytecode, it can be performed on-the-fly and needs no permanent modifications to third-party codebases and it is simply much easier to support new sources and sinks. How the instrumentation is done in detail will be covered by section 4.4.

Beside performing this as a preprocessing step, as done by Haldar et al. [4] as well as by Bell and Kaiser [31], this can be done on-the-fly using another command line option of modern JVMs. The standardized `-javaagent` parameter can be used to hook multiple, so called, *Java agents* into the class loading process of the JVM. In a nutshell, they are provided with functionality to register "transformers"[8], which are able to view and modify the bytecode of every classfile loaded after the JVM finished bootstrapping.

By this, the system is able to transiently instrument any classfile loaded into the JVM – without worrying about the classpath or custom classloaders of a given application. The approach is also covering special cases as encrypted classfiles[9] or ones retrieved by a network classloader (e.g., via `java.net.URLClassLoader`).

In combination, source code augmentation and bytecode instrumentation can be

[8] A transformer is an implementation of the `ClassFileTransformer` interface; `https://docs.oracle.com/javase/7/docs/api/java/lang/instrument/Instrumentation.html`

[9] This idea of encrypting classfiles and then decrypting it during classloading is quite popular in the community, although it is basically useless as any transformer hooked into the classloading process behind the decrypting one would be able to dump the unencrypted bytecode.

```
❯ java \
-javaagent:juturna.jar=taint-config.json \
-Xbootclasspath/p:juturna.jar \
-jar app.jar [optional parameters for actual application]
```

Figure 3.2.: This example shows, how the `-javaagent` and `-Xbootclasspath` parameters are used to run a Java application in the taint-aware environment provided by Juturna.

used to craft a system fulfilling the requirements stated initially. An example of how a JVM needs to be started in order to setup Juturna's taint-aware environment is shown in figure 3.2.

3.5. Defining the taint policy

Schwartz et al. [32] split the concept of a "dynamic taint policy" into three major parts: taint introduction, taint propagation and taint checking. The work at hand is going to follow their example.

3.5.1. Introducing taint information in a system

In order to have a taint flow, there always has to be a taint source. In Juturna there are, on a conceptional level, two classes of taint sources: implicit and explicit ones. Explicit sources are the ones defined by a developer/security expert, as part of the taint configuration when preparing Juturna for a given application, by listing their FQNs. They are completely under the configurators control. Currently, only methods returning a string-like type are able to be declared as source in Juturna.

On the other hand, there are implicit sources directly integrated into Juturna, respectively the standard library. An example for these is `AbstractStringBuilder#setCharAt(int index, char c)`, updating the character contained in a `String` instance at a a given index with another one. Due to the fact that this character "has no history", it might be, expecting the worst, considered suspicious. Another examples is a constructor in the `String` class receiving a `char[]`.

It needs further evaluation whether these implicit source are actually useful for the spotting of suspicious behavior trying to circumvent the tracking system, or whether they just create to much noise as they could lead to false-positives.

To give users a more powerful tool in general and to avoid the false-positives otherwise introduced via those implicit sources, there is another capability that distinguishes Juturna from other implementations. Juturna provides the ability to distinguish between different, user defined categories of sources. Every source in

an application is assigned to such a category. Additionally, Juturna allows these categories to be assigned to one of three levels. This will become clearer when having a look at the implementation in section 4.2.1 later.

- **`ACTUAL_SOURCE`:** For explicit taint sources
- **`POTENTIAL_LAUNDRY`:** For implicit sources, debugging and analyzing
- **`SANITIZATION_FUNCTION`:** Values sanitized by a sanitization function are explicitly marked with taint instead of just removing the taint information. As covered in section 4.4.1, handling of taint sources and sanitization functions can be realized as one on the implementation level. This additional taint level exists in order to still be able to distinguish between a value originating from an actual taint source or from a sanitization function. The reason for this is pointed out subsequently.

Positive tainting

In 2006, Halfond et al. [14] introduced the, according to them, novel idea of "positive tainting" as an alternative to the traditional "negative tainting". The idea is simple: instead of marking untrusted content, trusted content is marked. By this, the scenario of loosing taint does not lead to a false-negative (a successful, but not detected attack). It rather becomes a false-positive, because it would have lost its mark telling that it is trusted content.

Halfond et al. claim that it is easier to annotate trusted sources than untrusted ones – this is for sure valid in their scenario, the detection of SQL injection attacks, but probably not in general.

Still, Juturna is able to also support this strategy by simply marking trusted origins as sources, validation methods as sanitization functions and string literals as trusted by default as they are always considered to safe – the same as with negative tainting.

3.5.2. Taint propagation & semantics of strings in Java

The fundamental semantics, the propagation of taint information inside methods provided on `java.lang.String`, `java.lang.StringBuilder` and `java.lang.StringBuilder` should follow, are pretty straight-forward – at least on a conceptional level.

We will subsequently have a look at how the two primitives – which basically all modifying string operations can be reduced to – propagate taint information in Juturna. A merge of two different taint information is not possible, as we track taint with the highest possible resolution (character-level) – which prevents any loss of precision.

In the following the formal parameters `a` and `b` are strings, `s` and `e` are integers representing a start and an exclusive end index. `c` shall be understood as the resulting string. We assume that the taint information is represented by an array that can be accessed via a field named `taint`. The actual implementation is a different one and will be introduced in section 3.6.1, but for better understandability we forget about this for the moment. The following, very simple formulas show how the taint information is copied on an semantic level. It happens analogous to the copying of the character data.

- **`c := append(a, b):`**

$$c.taint[i] := a.taint[i]; \forall i \in \{0, 1, ..., length(a) - 1\}$$
$$c.taint[j + length(a)] := b.taint[j]; \forall j \in \{0, 1, ..., length(b) - 1\}$$

- **`c := substring(a, s, e):`**

$$c.taint[l] := a.taint[l]; \forall l \in \{s, s + 1, ..., e - 1\}$$

Beside this basic taint propagation policy, there are some Java specific string characteristics that influence the propagation policy. One should be aware of them, as they are important for the implementation of taint tracking in Java. Some more snares will be mentioned in the implementation chapter. In the following, the outline will stick closely to the one used by Chin and Wagner [40] while introducing some background information for enhanced comprehensibility.

Equality of strings & `hashCode()`

In Java, two instances – `a` and `b` – of any object, including all string-like types, are equal if `a.equals(b)` yields true. In case of the string-like types, this check is performed by comparing the internal character arrays.

Closely related to equality is the `hashCode()` method, which is also defined in `java.lang.Object`. It calculates an integer used for fast comparisons of objects and is usually overridden to fit the respective class. If two objects are equal, their hash values have to be identical – but identical hash values does not necessarily mean the objects are equal. To be equal, two objects do not have to be the same ("==", two references pointing to the same object).

But what happens if two actual equal strings, but with different taint information attached, are compared? Including the taint information into the comparison would break existing logic, therefore taint state must not influence those methods.

String literals & Java's string pool

The usage of *constant pools* is a technique implemented in the JVM and Java compilers to reduce the amount of memory needed for storing constants in a classfile and at

runtime. They work like symbol tables, e.g., known from compiler construction, and ensure that identical string literals are represented by just one `java.lang.String` instance. The compiler puts literals and strings computed by constant expressions into the constant pool of the classfile. When loaded by the JVM, they get transferred into its runtime string pool. As stated in the Java Language Specification [54, section 3.10.5], the process of storing string literals in the string pool is called *interning*. This interning can also be triggered at runtime by calling `intern()` on a given string. The method returns a reference to an equal, see above, instance of `java.lang.String` stored in the pool, or to the instance the method was called on in case there is no equal string in the pool yet. A common pattern to make a string "unique" therefore is `a = a.intern()`.

The problem arising from this clever mechanism is illustrated in the following code listing. Assuming, we have two identical instances of `java.lang.String`, `a` and `b`, in terms of the character sequence contained. The former is tainted, the latter is not. There is no equal string in the pool right now.

```
1  // Explicit usage of the constructor is necessary and will cause a char-wise copy to be
   created.
2  // Otherwise, a and b would point to the same instance in the string pool.
3  String a = new String("foobar");
4  String b = new String("foo" + "bar"); // Literals get combined at compile time
5
6  taint(a);
7
8  assert(a != b)
9
10 // Variant 1
11 a = a.intern();
12 b = b.intern();
13
14 assert(a == b)
15 assert(isTainted(b));
16 // => b now references to the same instance as a, which is a tainted string. This could
   lead to a false-positive.
17
18 // Variant 2, run instead of the first one
19 b = b.intern();
20 a = a.intern();
21
22 assert(a == b)
23 assert(isTainted(b) == false);
24 // => a now references an untainted string, the same as b points to. This could lead to a
   false-negative.
```

Listing 3.3.: A Java snippet illustrating the problems caused by `String#intern()`.

As shown in listing 3.3, interning can now lead to the loss of taint information, leading to a false-negative, by replacing a tainted string with an untainted one – or vice versa resulting in a false-positive then. This behavior possibly decreases precision and recall, but cannot be prevented while not touching the semantics of Java's strings.

Including a check in `intern()` preventing it to return another instance in case of differing taint information would not harm backward compatibility – but is technically impossible as the method is natively implemented (and dynamically linked) and therefore just declared, but not defined, in `String.java`. As `intern()` internally might call `equals()`, a horrible workaround could be to modify this in order to perform the additional check, but only when – according to the stack trace – called from `intern()`.

The probability of this problem to occur in real-world examples is, admittedly, very low as `intern()` is rarely explicitly used and, in order to remove taint information from a string, an identical one without taint attached would have to be in the pool already. Chin and Wagner [40] share the estimation that this is not a feasible attack vector. Still, during implementation and testing this led to confusing errors.

As `java.lang.StringBuilder` and `java.lang.StringBuffer` are not "pooled", this issue does not arise in these cases.

Immutability

`java.lang.String` instances are immutable, i.e., every modification to a string causes a new instance to be created. Correspondingly, methods modifying a string cause the respective taint information to be attached to the newly created string. The taint information of the original string does not get altered.

Serialization

Java Object Serialization is a mechanism for persisting and restoring an applications state. It offers automatic, or explicitly implemented, handling of accordingly marked classes in order to generate a representing byte stream of primitives. All of the three string-like types implement the `java.io.Serializable` interface and therefore are serializable. This interface has no fields or methods, it is simply used as a signaling interface so the runtime knows that is is legit to serialize a given class.

As serializing the taint information as part of, e.g., a `java.lang.String` instance's internal state would lead to a dump not deserializable/restorable in a runtime not using Juturna's augmented standard library – breaking compatibility with those environments.

On the other side, not serializing taint information results in another possibility to bypass the system: serializing and subsequent deserializing string-like types would vanish the taint. But this can be considered malice.

Still, it seems more important to prevent loosing taint than ensuring interoperability between taint-aware and non-taint-aware Java runtimes – but changing the way Java serializes instances of `java.lang.String` is not possible via the standard mechanism (overwriting `writeObject()` and `readObject()`) as the "class String is

special cased within the Serialization Stream Protocol"[10].

`java.lang.StringBuffer` and `java.lang.StringBuilder` are handled via the default mechanism, but changing them without `java.lang.String` would result in breaking compatibility without actually fixing the issue of taint loss.

Ultimately, Juturna does not serialize the metadata it tracks and considers code using serialization to be checked by a developer/security expert.

Encodings & locales

Java uses the character set described by the Unicode standard. Unicode pursues the goal of containing all characters used in the world and assigns each of them a code point, ranging from 0x0 to 0x10FFFF.

Unicode itself is not an encoding, this is provided by, e.g., *UTF-8* and *UTF-16*. Java uses UTF-16 which is optimized for encoding most characters used in the western world with one 1 *code unit*. Ones with a higher code point need 2 code units and are called *supplementary characters*. Code units are represented by the primitive `char` type in Java, a 16 bit value. Therefore, all western characters can be encoded with one `char`, but Asian ones might need two – so the "character" in character-level granularity actually relates to `char` values in the implementation, not to actual glyphs and letters.

Due to this "content-agnostic" representation, it is possible to have a glyph represented by two code units/`char`s, one of them being tainted, the other one not. Chin and Wagner consider a supplementary character to be tainted in case one of its code units is tainted. As Java itself does not do special handling of such supplementary characters (e.g., there are no checks that `substring()` or `replace()` always cover all code units of such a glyph), Juturna does neither.

Nevertheless, awareness of the internal representation of glyphs is important as there are special cases for some locales. For example, calling `String#toUpperCase()` on a string containing "ß" results in "SS"[11]. By this, another character has been added to the returned string – requiring to be given the same taint. Additionally, the taint information of all subsequent characters in this string has to be shifted. Another example, and handling of this, will be described in the regarding implementation section (section 4.3.2).

A simply bytecode instrumentation would not be able to handle these situations, as there is no explicit relation between the lowercase and uppercase glyph. But with the (manual) augmentation approach, such implicit relations can handled. One could argument, that the uppercase characters are to be trusted and not tainting them would be consequent. On the other side, from the semantic point of view, it seems obvious that they should be tainted.

As there seem to be no such special cases shrinking a glyph represented by 2 code units down to a single one, merging of taint information cannot occur.

[10] Comment taken from `java.lang.String`

[11] This obviously does not take into account that, just recently, a capital version has been introduced into the German language.

Primitive characters (`char`)

As explained before, the taint tracking mechanism implemented by Juturna is not capable of tracking primitive `char` values when they are standing on their one and not contained in a string-like type.

This leads to conceptional problems with methods like `AbstractStringBuilder#setCharAt(int i, char c)` and `...#replace(char oldChar, char newChar)` as they replace a possibly tainted character with one "without history". Juturna faces this by using implicit sources as described above in section 3.5.1. By this the inserted `char` is always tainted. This behavior differs from the one of Chin and Wagner's system which simply does nothing, therefore keeping the taint state of the replaced character.

3.5.3. Taint checking, untainting & sanitization

Checking for taint information is done at sinks, which have to be methods receiving string-like types as parameters. It is defined by its FQN and the index of the respective parameter. Additionally, a list of forbidden kinds of taint sources and a mitigation strategy can be given[12].

We consider a vulnerability to be found in case the defined parameter contains at least *one* character originating from a source contained in the list of forbidden sources.

A detected taint flow, from the semantic point of view, does not necessarily mean that a real attack has been detected as the string flowing into the sink could be benign in this context. Just think of a valid name, not containing illegal characters, embedded into a SQL query template and then passed to a database driver. But deciding whether the strings content is harmful in a certain situation can be considered to be impossible to determine in general. Therefore, in the context of this work, the problem of detecting an (injection) attack will be reduced to the detection of a tainted string flowing into a sink.

In such a case the defined mitigation strategy gets executed. If *all* tainted characters are linked to the `SANITIZATION_FUNCTION` severity level, then the taint check is positive.

Untainting is necessary in order to remove taint from a system, preventing the number of tainted information to be monotonously increasing. Systems as the one proposed by Chin and Wagner [40], using just a binary flag, obviously need to realize the operation of untainting by completely removing the taint information. As Juturna is capable of tracking 2^{16} different sources, an alternative strategy has been chosen: sanitization functions are basically the same as taint sinks, but the taint attached by them is of level `SANITIZATION_FUNCTION`. Juturna never removes taint information by itself – taint information can only be overridden, possible causing taint loss to strike more attention. By this, Juturna actually integrates the

[12]For details, please see section 4.4.2.

aforementioned idea of positive tainting as introduced by Halfond et al. [14].

Unlike other approaches Juturna does not implicitly untaint data, as a strategy/heuristic for determining such situations seems to be very hard to define. The heuristic used by Haldar et al. is a good example for this, as it is very simple and very prone to result in false-negatives: they simply assume "that methods of java.lang.String that perform checking and matching operations are used to untaint strings" [4] as they expect those to be part of sanitization operations.

3.6. Ideas towards more efficient taint tracking

3.6.1. "Taint ranges" for more efficient storing of taint information

So far, an array-like data structure, added to the string-like types in form of a field, has been assumed for storing attached taint information. All string-like types[13] internally use an array of type `char` to store the UTF-16 code units of the sequence of glyphs they are representing. Usage of another array for the taint information therefore seems natural – and is indeed very popular: Chin and Wagner [40] use a `boolean` array, Bell and Kaiser [31] one containing `integer` values or strings – depending on the mode their system Phosphor is configured to run in.

This results in linear space complexity as there is a 1-to-1 relation between the elements in the `char`-array and in the "shadow array" containing the taint information. For `java.lang.String` instances the computational overhead is also linearly linked to the amount of code units spanned as those objects are immutable and a copy needs to be created – even for the smallest modifications. In case of the two remaining string-like types, which are not immutable, problems arise when the character sequence gets longer as arrays are of fixed size in Java. Sticking to arrays therefore results in the need to replace the array with a new one and to transfer the information. In case of not changing the strings size, computational effort solely depends on the modification performed.

Due to these downside, this work proposes the usage of an enhanced data structure called "taint ranges" – a novel approach in the world of taint tracking for Java.

Juturna does not use such shadow arrays, instead every string-like type basically has an additional list of quadruplets. Each of them contains start and end of a range plus a value indicating the kind of taint source and an extra field for debug-information. The cooperating SAP department already implemented this idea for taint tracking systems supporting other platforms, e.g., the JavaScript engine of Mozilla Firefox [55]. It is loosely based on the idea of a *Run-length Encoding* (RLE).

As we will see in section 4.2, these taint ranges are implemented as immutable objects. Therefore a derived string can simply reuse ranges of the original string by adding them to its own list (as references, no copy needed).

[13] `java.lang.String`, `java.lang.StringBuilder` and `java.lang.StringBuffer`

We assume the average string flowing through an applications not to contain characters originating from more than a few different kind of taint sources – probably just a one in most cases. The length of the average, user-controlled strings are considered to be quite short – but may grow massively when getting combined with (HTML) templates and the like. For long character sequences, the concept of taint ranges pays off well, as Juturna does not need an array of length n to store taint information for a string of length n. Containing only characters from one source, it exemplarily needs just one range with constant memory consumption. This constant size is low enough to not noticeably increase the memory consumption for short strings. We will discuss this in more depth in the implementation section – after going through internals like datatypes used, which need to be set before comparing to the "naive", shadow array approach.

The computational overhead might be a little higher compared to the naive variant, as the operations necessary on taint ranges (e.g., insertion of a new range leading to the split-up of an existing one) are more complex.

3.6.2. Using static IFC to make taint tracking more efficient

Dynamic taint tracking adds overhead due to additional operations performed at runtime in order to keep track of explicit flows. These additional operations might be performed more efficiently – but the overhead in general cannot be avoided as this is an immanent characteristic of such a dynamic analysis.

The only way to avoid the overhead is to avoid the analysis – the question to be answered must therefore be, whether it is actually necessary to perform the analysis at all times.

Looking at the concept right now, additional operations are always performed – no matter whether a string containing attacker-controlled input, i.e., originating from a source, actually reaches a sink or not.

For strings not originating from a source, there currently is (almost) no additional, computational overhead and no memory consumption[14].

But consider a string originating from a source and not flowing into a sink in the end. Because the string is indeed tainted, propagation needs to be performed – therefore, causing computational overhead and allocation of memory without being checked at a sink in the end. In this case, tracking taint information and keeping it in sync is actually superfluous as this string's taint state never gets evaluated due to not flowing into a sink.

However, the prediction whether a tainted string might, in the end, flow into sensitive areas of an application or not cannot be made in advance and therefore propagation of taint information can just be proven unnecessary after reaching a variable's lifetime. In many cases, e.g., with (static) fields/members, this will be

[14] To be more precise, there is a simple runtime check whether a string is carrying taint information or whether propagation can be skipped.

pretty much equal to the end of an application's execution – at least with dynamic approaches.

But static analysis techniques are able to compute these irrelevant flows in advance. Therefore, this thesis tries to combine mechanisms from static *Information Flow Control* (IFC) with dynamic taint tracking, selectively avoiding propagation of taint information – and also avoiding its intrinsic overhead – for information flows that can be guaranteed to not be security-sensitive, i.e., not flowing into a sink nor influencing another value doing so.

This idea of "selective taint tracking" is realized by some more source-level additions to the standard classes, by adding another on-the-fly bytecode manipulation and by the application of PDG-based IFC mechanisms in a preprocessing step in order to find the parts of an application not necessarily needing to be taint-aware. As aforesaid, the three string-like classes have been modified in a way that all methods manipulating the contained character sequence also analogously modify the attached taint information.

In order to avoid this additional code to be executed, the source code of the initially included, unmodified methods get added to these classes again. Then, during classloading, all method calls guaranteed to only be invoked on instances of string-like types being considered "safe" are replaced by calls to the taint-unaware original methods.

A method call describes a single invocation of a method, e.g., `a.foo()` would be an invocation of the method `foo` on all instances the variable `a` might point to.

In the context of an instance/object, "safe" means that it does either not origin from a taint source, is not influenced by another doing so or simply does not flow into a taint sink. A method call/invocation is considered to be safe only when all instances/objects possibly assigned to the variable/identifier/designator in operates on[15] are guaranteed to be safe. This corresponds to IFC's concept of integrity, declaring an information trusted when not influenced by an untrusted one.

This might be trivial to decide in case of a locally defined variable pointing to a newly created `String` – but as soon as this variable is declared as a formal parameter of a function, possibly called from various parts of the application assigning many different instances, deciding (correctly) is far from trivial.

But this is not a new problem in computer science and modern static IFC mechanisms are capable of solving this problem, guaranteeing correctness (and therefore "safety").

Before discussing how the safe method calls can be determined, figure 3.3 tries to visualize the idea of tracking taint only selectively.

[15] Static methods are not of interest in this approach, as they should not be able to operate directly on internals of a given instance (due to encapsulation and the concepts of visibility) and therefore do not need to perform taint propagation.

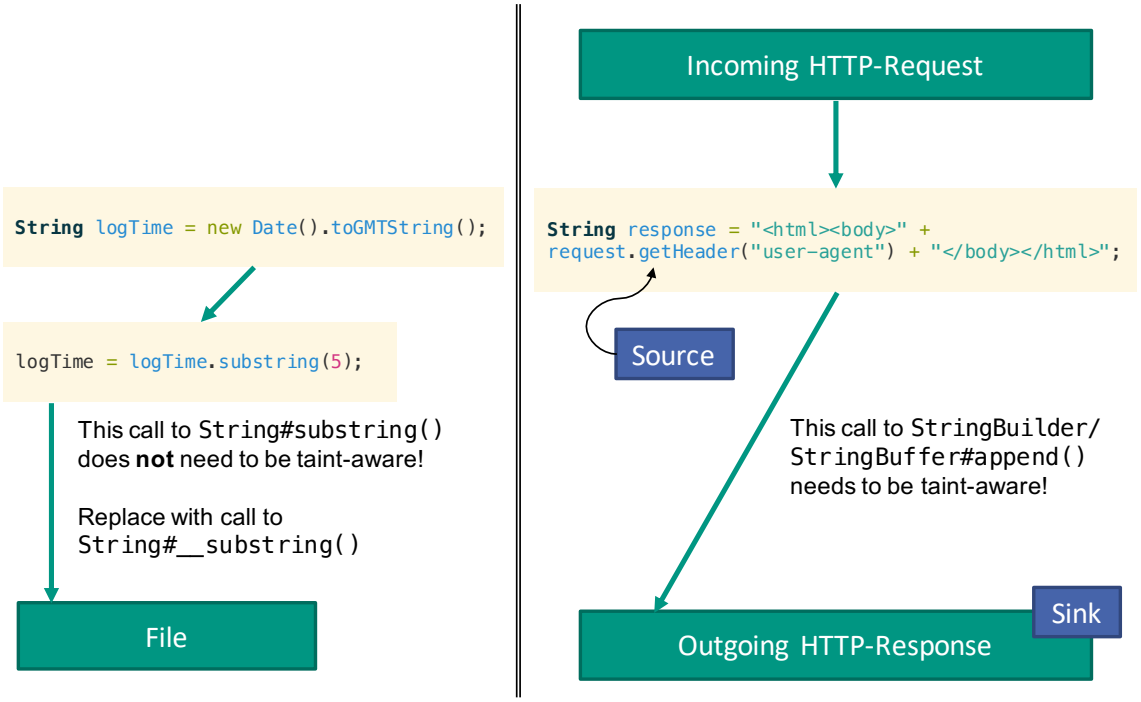

Figure 3.3.: An example showing the idea of "selective taint tracking". On the left, a string not tainted and not flowing into a sink makes taint tracking superfluous. On the right, a tainted string returned from a source (`...#getHeader()`) gets concatenated with a literal. The Java compiler compiles this code to use a `StringBuilder` for concatenation. As the resulting string ends up in a sink, `StringBuilder#append()` needs to be taint-aware.

This is where *JOANA-Adapter* comes into play. It is a small tool, whose design and implementation are essential parts of this work, scanning a given applications bytecode for safe method calls as a preprocessing step. As the name implies, it uses the aforementioned JOANA library internally, providing PDG-based IFC analysis functionality.

Operating on a *System Dependence Graph* (SDG), the tool computes, starting from the methods defined as sources, forward slices containing the parts of an application that might be influenced by/depending on a tainted value. These parts, represented as nodes in the SDG, are merged into $S_{forward}$. Additionally, backward slices are computed starting at the defined sinks containing all nodes influencing them/they are depending on ($S_{backward}$).

The intersection $S_{intersection}$ of these two sets $S_{forward}$ and $S_{backward}$ represents the parts/nodes of an analyzed application potentially being "unsafe" according the definition given above. Method calls operating possibly on at least one instances contained in $S_{intersection}$ need to be taint-aware and are not safe. The remaining invocations, not being contained in $S_{intersection}$, are considered safe and will therefore be enqueued for being replaced with a taint-unaware variant.

This described operation is very similar to the more precise concept of *chopping*, which has been introduced by Jackson and Rollins [56][16]. For simplicity, the approach of computing the intersection is used in the explanation and the upcoming example – although the actual implementation performed later on uses chopping.

Listing 3.4 illustrates the idea by using pseudocode.

[16] An enhanced, more precise algorithm for inter-procedural chopping has been presented later on by Reps and Rosay [57].

Data: $G_{application}$, abstract graph containing abstract nodes and edges representing a given application; $S_{sources}$, set indicating which abstract nodes in $G_{application}$ are sources; S_{sinks}, set indicating which abstract nodes in $G_{application}$ are sinks

$$
\begin{aligned}
S_{forward} &\leftarrow \bigcup_{source \in S_{source}} \texttt{forwardSlice}(G_{application}, source) \\
S_{backward} &\leftarrow \bigcup_{sink \in S_{sink}} \texttt{backwardSlice}(G_{application}, sink) \\
S_{not\_safe} &\leftarrow S_{backward} \cap S_{forward} \\
S_{safe} &\leftarrow G_{application} \setminus S_{not\_safe}
\end{aligned}
$$

Listing 3.4.: Pseudocode showing the basic idea behind the presented "selective taint tracking" approach. Parts of an application considered "safe" are determined.

With this information at hand, a bytecode instrumentation, as used for introducing taint sources and sinks into a given application, is able to exchange safe method calls with ones to unaugmented/taint-unaware versions.

Switching to the unaugmented methods – and not vice versa – has one big advantage: it is ensured that unprocessed applications only call taint-aware functionality and usage of Java Reflection for invoking methods by name at runtime does not circumvent the protection as the augmentation is not performed on caller-site. In case the bytecode rewriting fails, or is not sound, this results in unnecessarily executed tracking code instead of loosing track due to calling a taint-unaware method. Thus, the system follows the common "secure by default" pattern.

As discussed in the section on related works (section 2.5), the basic idea of combining static and dynamic analysis is not a new approach. But the way they are combined in this work is – as far as the author is aware of after meticulous search – unique (in the Java world).

Section 4.5 dives deeper into the actual realization of the various components involved and their current limitations.

4. Implementation

After extensively discussing the underlying ideas and concepts of Juturna, and taint tracking in general, in the last chapter, this one will dive into the details of the implementation part of the thesis. As discussing every single aspect of the implementation would be unrewarding and go beyond the scope of this document, the focus will be put on the more interesting and more complicated parts implementing the conceptional main points.

The implementation has been primarily done in *Kotlin*[1], a statically-typed language based on the concepts of Java, but with a more lightweight syntax and handy features like *companion objects* enabling inheritance for static methods and fields, optional parameters with default values and an enhanced type-system helping to avoid `NullPointerException`s. It further encourages developers to use patterns known from the world of functional programming more often.

The language has been developed by *JetBrains* and can be compiled into Java and *Dalvik* bytecode, JavaScript and even native machine code for the major platforms. Interoperability between Java and Kotlin works great as the latter borrows the main part of its standard library from the former. Therefore, it is easy to mix Java and Kotlin code in a single application.

Modifications to the standard library have been done in Java in order to keep their impact as low as possible. All additions to existing classes have been explicitly marked (`//taint>` and `//<taint`) in order for them to be easily recognizable. Additionally, they are tracked by *Git* as the *Version Control System* (VCS) of choice.

In the next sections, several *UML* diagrams will be presented. While creating them, a lot of attention has been paid towards keeping the focus on important aspects and avoiding details which are not necessary in order to reach a certain level of understanding. Therefore, private fields and methods are not shown. The same is true for "non-constructive" public ones, e.g., redefinitions of `equals()`, `hashCode()` or `toString()`.

During the implementation phase attention has been paid to follow common best-practices in software engineering as, e.g., postulated by Robert C. Martin in *Clean Code* [58]. Whenever it made sense, common patterns have been applied. As the system shall strive for a reasonable performance, optimized data structures like linked and/or hashed lists, maps and sets have been used together with algorithms exhibiting good runtime and space requirements.

Implementation and testing has been done using Oracle's JDK 8. The single

[1] `https://kotlinlang.org/`

modules use `Apache Maven` as build system, some examples are handled by `Apache Ant`, `make` or *shell* scripts. As far as possible, tasks have been tried to be automated.

Huge effort has been put into implementing Juturna thoroughly and into the following sections describing its functionality – in the hope that it might be reused.

4.1. Overview of components

The implementation consists of many different (logical) components. They can be grouped into the core components, laying the foundation for the taint tracking system, and additional ones, enhancing the system (*JOANA-Adapter*) or being used for the purpose of evaluating the system (*Juturna-Benchmark* and the example applications).

On the source code level the core components make up one big Maven module, *juturna-core*, beside *juturna-benchmark* and *JOANA-Adapter*.

The idea of splitting up a project in smaller, more maintainable components should be considered a best-practice in the field of software engineering. Although, not all components are clearly separated regarding the codebase, it still seems appropriate to talk of components in order to give the following sections, describing the core components shown in figure 4.1, a better structure.

There is a dedicated section for the JOANA-Adapter (see section 4.5). *Juturna-Benchmark* and the example applications will be discussed in the evaluation chapter.

In order to get a rough overview on what the core components' objectives are, they are briefly described:

- **Taint-Storage:** Provides functionality for storing the taint information. Contains classes for representing and managing taint ranges in an efficient manner.
- **Augmented standard library classes:** Contains the classes taken from the standard library that have been augmented in order to be capable of storing and propagating taint information during execution. At runtime, these modified classes are used instead of the ones contained in the JRE and therefore provide a taint-aware foundation.
- **Java-Agent:** This component is responsible for instrumenting the classfiles loaded into the JVM. It uses standard mechanisms to hook into the class-loading process and applies multiple transformers on the loaded bytecode in order to mark methods as sinks and sources. This component also contains the instrumentation necessary for providing selective taint tracking.
- **Taint-Check:** Among others, the Java-Agent adds code for the operation of taint checking to methods marked as sinks, actually simply calling the sophisticated checking functionality implemented in this component.
- **THelper:** The class `THelper` is used to access taint information attached to a taint-aware type at runtime. Trying to access these information from inside an

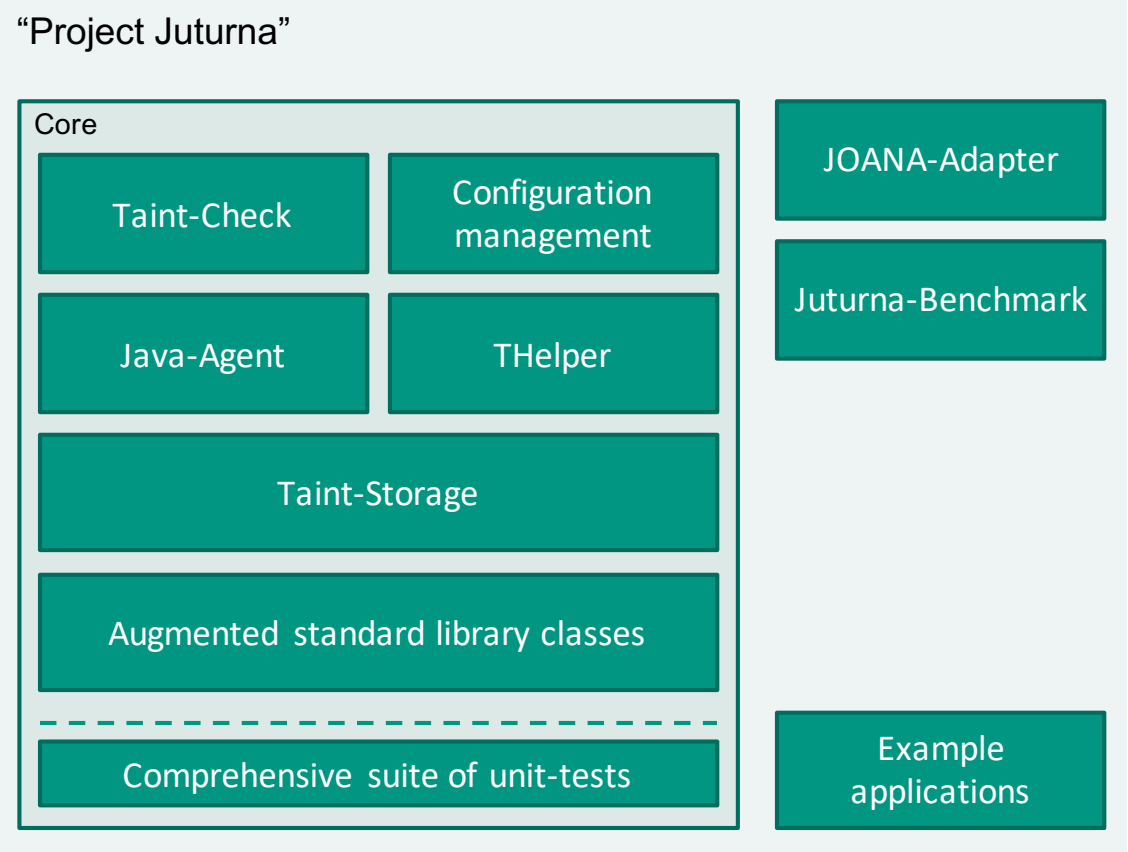

Figure 4.1.: The various components of Juturna

application might be tricky, as the Java compiler will insist, that the according fields and methods do not exist on `String` and the others as it usually is not aware of the augmented standard library classes. By using Reflection inside `THelper`, these problems can be circumvented. It is massively used in the test-suite.

- **Configuration management:** The Java-Agent can be configured to mark arbitrary methods as sources and sinks. Additionally, the different kinds of taint sources need to be introduced and selective taint tracking can be set up. This component performs loading of configuration files and, as configurations are able to extend each other, possibly merges them.

4.2. Mapping metadata to strings: `TaintRange` and friends

The `com.sap.juturna.taintStorage` package (see figure 4.2) contains all classes needed in order to describe and manage taint information attached to one of the three string-like types.

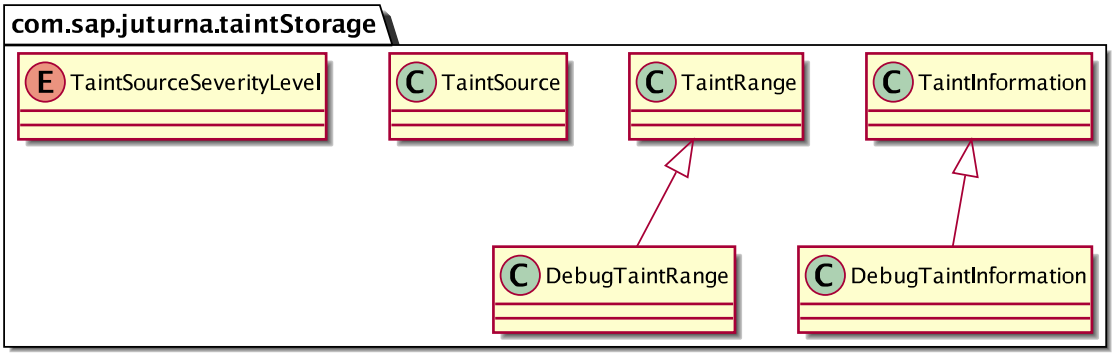

Figure 4.2.: Classes contained in package `sap.com.juturna.taintStorage`; they provide the foundation for storing taint information within the string-like types.

4.2.1. Structure of a taint range

An instance of `TaintRange` basically just combines three numbers: an `int`s for `start` and `end` each, plus a `short` representing the taint source type, i.e., which kind of taint source, the range originates from. The string-like types in Java internally hold an array of `char`s, representing the UTF-16 code units. As addressing items in an array is done using integers, a string cannot be longer than `Integer.MAX_VALUE`[2] – `int` therefore is the suitable primitive. The type `short` offers only half the amount of bytes, enabling the encoding of 2^{16} unique kinds of taint sources (or 2^16 unique source, as every source could be of a different category).

`start` and `end` are both absolute values. The decision not to use relative values, or an absolute start and a field indicating the length, will be explained in section 4.2.2 when going into the methods operating on taint ranges.

Instances of `TaintRange` are immutable, meaning they cannot be modified anymore after having been created. Immutability is a popular concept in order to avoid problems arising by the usage of multiple threads operating on shared objects as

[2] Actually, this is just an upperbound as it depends on the JVM in use and the real limits seems to be below this value usually. In `java.lang.AbstractStringBuilder`, one can find a comment stating that Integer.MAX_VALUE − 8 is the maximum size for an array as “some VMs reserve some header words in an array”.

harming sequential integrity is not possible without write-operations. Furthermore, it allows an object to be safely referenced by different parts of an application, without creating dependencies between them and the risk of accidentally having one part changing the data used by the other.

`TaintRange` instances are not directly attached to string-like objects, instead they are managed by instances of `TaintInformation`. Because of their immutability, taint ranges can be shared by multiple of these containers – fulfilling the *flyweight pattern* by minimizing memory usage through sharing. This especially comes into play when, e.g., appending one string to another as the ranges of the first one can simply be reused. If `start` and `end` where relative and not absolute indices, the ranges of the second string could also be reused[3]. Still, the implementation uses absolute values as relative indices have other disadvantages.

The implementations of `TaintInformation` and the augmented methods try to reuse ranges whenever reasonable, but this is no must. There is no global "uniqueing" of taint ranges, so there might be multiple taint ranges for which `equal()` would yield true.

Beside `TaintRange`, there is also a class `DebugTaintRange` which inherits from the first. Its purpose is to additionally store the current stack trace when setting the taint information in order to help with debugging and finding the origin of tainted information. The decision, which of these two to instantiate, is resolved based on a central configuration and is transparent for a developer using taint ranges when adjusting further system classes or libraries. We will come back to this when looking at `TaintInformation` and `DebugTaintInformation`.

`TaintSource` keeps track of the categories of sources internally and therefore provides a static function, similar to the well-known *creator pattern*. They name might be a little misleading, as it does not refer to a source, i.e., a method returning strings considered tainted, but to the category this source might be assigned to. Therefore, a `TaintSource`, e.g., named "HttpEntrypoint", can refer to a whole set of sources.

The first category of sources is assigned the id -32768. In order to use the full range of `short`, while still being able to provide a fast, array-based lookup by an id, it is treated as an `integer` internally and an offset is added to make the id -32768 point to the array element at index 0. Fast lookup by name is realized using a `HashMap`.

Beside an id and a name, a `TaintSource` instance has a constant from the `TaintSourceSeverityLevel` enum assigned. This is necessary to distinguish between different levels of functions emitting taint, as described in section 3.5.1 before. There are three levels defined in `TaintSourceSeverityLevel`: `ACTUAL_SOURCE`, `POTENTIAL_LAUNDRY` and `SANITIZATION_FUNCTION`.

[3]Except the first one as it might need to be adjusted.

4.2.2. TaintInformation: a container for taint ranges

Instances of `TaintInformation` provide a container for taint ranges and functionality for handling them: fast lookup, insertion, appending, adjusting, etc..

Same as with the taint ranges, there is a "debug-pendant": `DebugTaintInformation`. When an instance of one of them is needed, a static factory method gets called, returning the appropriate one according to the system's configuration. A returned `TaintInformation` internally just creates instances of `TaintRange`, a `DebugTaintInformation` just ones of `DebugTaintRange`. As taint ranges must not be created using there constructor, there is no risk of getting them intermixed.

The ranges are stored in an explicitly ordered `List`, although an implicitly ordered structure, like a `TreeSet`, might seem more suitable on the first sight. But as it is not possible to guarantee that a given taint range fits into a list of existing ones, e.g., in case of partial overlaps, this cannot be implemented (easily) with the `Comparator` mechanism used by Java. Instead, a standard list is used in conjunction with methods keeping the list sorted when inserting and exploiting the *total order* during lookups.

This is centralized in one method using *binary search* to determine the array index at which an item needs to be inserted/can be found in. This can be performed in, at maximum, $\log_2 n$ steps. Using binary search is only possible because taint ranges have absolute indices. Most of the methods in `TaintInformation` are able to perform faster using absolute indices – even so using relative indices would increase the reusability of the immutable taint ranges (as there would be less need to modify them) and therefore avoid copying in many cases.

Inserting a range into an area already covered by other ranges requires more sophisticated handling as there are, in general, several ways in which ranges can (partially) overlap. In case they do, the older one needs to be adjusted, or to be deleted when it is contained in the other. Adjacent ranges get merged into a big one in case the sources having emitted them are assigned to the same taint source category.

`TaintInformation` is not immutable and therefore needs explicit, internal synchronization in order to ensure that accessing one instance from different threads does not harm sequential integrity in case getting attached to further types in the future. These problems only arise when directly accessing the taint information attached to an object, as `java.lang.String` is immutable and modifications cause the creation of a new instance (with a new `TaintInformation` instance). `java.lang.StringBuffer` itself is synchronized already and `java.lang.StringBuilder` is explicitly declared not to be thread-safe.

4.2.3. Considerations regarding runtime and space requirements

To emphasize that space and runtime requirements were important aspects during the concept and implementation phases, this subsection will present some considerations regarding the asymptotic space requirements of taint ranges compared to

the former approach using shadow arrays. Regarding performance, the usage of optimized data structures and more sophisticated handling routines has already been mentioned – although this surely could be improved, but not in the timeframe of this thesis. A promising approach on further enhanced taint ranges will be presented in section 5.4.2, based on the insights gained during the evaluation.

The comparison baseline for these considerations will be the "naive approach" of storing taint information bound to a code unit as a number in an array, as done by Chin and Wagner [40]. Its space requirement r_{naive} solely depends on the size s_{naive} of the data type used to represent the taint source (category)[4] and the length n of a string: $r_{\text{naive}} = s_{\text{naive}} * n$.

It does not make a difference whether the string is actually tainted or not. In case the string is completely untainted, allocating the array can be avoided, as done by Chin and Wagner.

As we are looking at asymptotic approximations, we do not take the memory needed to store the reference to the array, etc., into account. We assume the smallest datatype, `byte`, to be used. So let s_{naive} be 1 byte[5].

Using taint ranges, the memory overhead, by contrast, solely depends on the number of ranges k assigned to a string, not necessarily related to the amount of characters. The length of these "blocks" is not of interest. Merging of ranges effectively limits k: $k \in \mathbb{Z} \cap [0, \lceil \frac{n}{2} \rceil]$.

A taint range is composed of 2 `integer` variables, each taking up 4 byte, and s 1 byte value indicating the source[6] – adding up to 9 byte. The space requirement can simply be approximated by $r_{\text{tr}} = 9 * k$.

As we see in figure 4.3, the space requirements of taint ranges are independent from the length of the string and k can be assumed to be quite low in real-world scenarios. Considering that k might correlate with n, taint ranges are more memory efficient as long as $\frac{n}{k} > 9$ holds true. Reusing taint ranges might additionally save memory.

The memory consumption of the string itself can be approximated with $r_{\text{string}} = n * 2$ byte as the `char` type has a size of 2 byte.

[4] Taint state in case of simple binary flagging.

[5] `boolean` might seem to need even less space, but that is a fallacy as popular JVMs internally represent `boolean` as a `byte`. Chin and Wagner use a `boolean[]` allowing them just binary flagging – but they probably could use `byte` instead and gain 2^8 distinguishable sources with the same memory footprint.

[6] Actually `short` is used, we assume `byte` for a fairer comparison. Nevertheless, usage of `byte` might be sufficient for most scenarios anyways and adaption can be done easily.

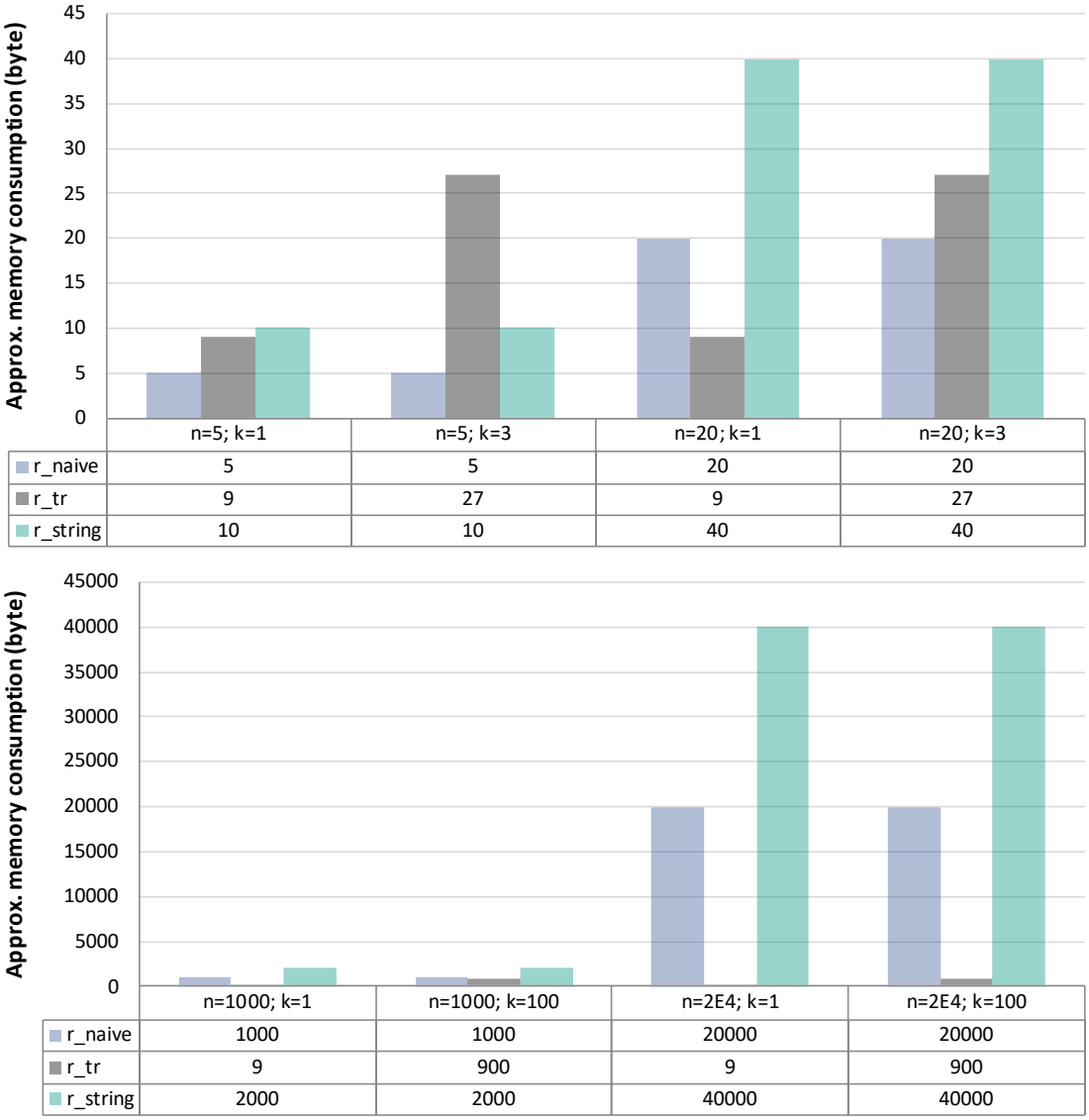

Figure 4.3.: The diagram shows an approximation of the memory required with a naive approach for storing taint information and for using taint ranges.

4.3. Augmenting the standard library

Augmenting the string-related classes contained in Java's standard library[7] was a major part of the implementation work done for this prototype. This was not only about adding code able to properly propagate taint information inside the various methods, it was also about understanding how these internals work and where adjustments would be needed.

Table 4.1 lists which classes have been adjusted and how many methods needed modifications. Together, these classes contain ≈ 7000 of code and documentation.

Class	Methods augmented	Methods in total*
java.lang.String	9	≈ 90
java.lang.AbstractStringBuilder	17	≈ 60
java.lang.StringBuilder	1	/
java.lang.StringBuffer	1	/
java.util.regex.Matcher	2	≈ 40

* Many of these are simple overloads setting default values or are, more-or-less, just delegating internally; takes constructors into account.

Table 4.1.: Classes of the standard library that have been augmented in order to be able to propagate taint information.

The process of augmentation started with scanning the three classes `java.lang.String`, `java.lang.StringBuilder` and `java.lang.StringBuffer` in order to find methods and further classes needing adjustment. Some methods are simple shortcuts, e.g., `String#subSequence(beginIndex, endIndex)` just delegates to `String#substring(beginIndex, endIndex)`), redirecting to other methods without containing any further logic. Others do not receive a string, or return none while also not modifying internal fields (like `String.charAt(index)`) – the remaining ones were good candidates for closer investigations.

The initial classes for augmentation have been taken from the *OpenJDK* project in the most recent version of OpenJDK 8[8].

The OpenJDK itself is licensed under the *GNU General Public License* (GPL) in Version 2, the standard library classes are additionally subject to the so called "Classpath" exception. The GPLv2 would consider a project including these classes, or derived versions, to be a combined work which would need to be licensed under GPLv2 too. The "Classpath" exception weakens this requirement by dispensing

[7] There is actually no common, coined term or proper name for the classes delivered with the JRE like they exist in `C/C++` world. Most refer to it as "standard library", others may use "Java class library".

[8] Link pointing to the exact revision in their versioning system: `http://hg.openjdk.java.net/jdk8u/jdk8u60/jdk/file/935758609767/src/share/classes/java/`

other parts of this combined work to be licensed under GPLv2. Solely the standard classes need to stick to this license, although the "Classpath" exception might be stripped. This probably is in the interest of SAP in order to prevent others from reusing this code without basically open-sourcing their whole system[9]. (cf. [59])

Still, the augmented standard classes are subject to the GPLv2 – but in case of just using Juturna internally or for hosted customer applications there is no obligation to provide the source code of these classes as the GPLv2 only requires source code to be shipped along with compiled code; in other words the idea behind GPL falls short with on-demand cloud services as there no piece of work is published or distributed. (cf. [60])

4.3.1. General considerations

There are some general aspects which apply to most of the touched classes similarly, such as how to attach the taint information to the three string-like types. The first step after understanding the internal structure of the string-like types was to add a field named `taint` of type `TaintInformation`. This has been done with `java.lang.String` and `java.lang.AbstractStringBuilder` which is the super class of `java.lang.StringBuilder` and `java.lang.StringBuffer`).

When creating a new instance of any of the string-like types, the `taint` field is initially assigned `null`. This has two advantages: In case a string never actually becomes tainted and therefore no `TaintInformation` instance is needed, instantiating can be avoided[10]. It is also a flag showing the overall taint state, i.e., whether the string is tainted or not, and is used in order to shortcut and skip propagation routines in case of the latter. The second advantage is much less obvious. Java uses pools containing constant values of which the aforementioned string pool is surely the most prominent one. When compiling Java source code, the compiler puts string literals found in the code into a specific section of the classfile, they then get transferred to the string pool when executing the file in a JVM. Therefore, these character sequences become instances of `java.lang.String` without actually being instantiated (!). This behavior was observed during tests with an additional field added to `String` assigned a non-null value – but containing just `null` when accessing it. The assignment, which has been moved to the special `<init>` method by the compiler, was never executed for these literals. Simply going without initialization of the `taint` field frees from needing to work around this.

In order to mark the string types as taint-aware, they now additionally implement the interface `java.lang.TaintAware`. Beside providing a common parent for the taint-aware types, this enforces them to have a method `isTainted()`. The `taint` field itself could not be introduced by the interface as Java as a language does not support this.

[9] The legal wording in the source code files has not been changed during the work on this project.

[10] Chin and Wagner [40] use this idea to avoid allocating arrays for the taint information if not really necessary – but with them, this is of much bigger impact as their `boolean[]` is not just an empty container.

For every touched method attention has been paid to not accidentally change their behavior – neither directly, nor indirectly by side effects caused by the added code. As the objective was to keep performance on a reasonable level, augmentation has been done with the context and purpose of the respective method in mind in order to handle the propagation of taint in a more efficient way, instead of trying to apply more or less the same pattern to all methods.

`equals()` and the serialization mechanism have not been touched, as discussed in section 3.5.2).

The way the standard library has been adjusted can be seen as a template for augmenting further classes and libraries.

4.3.2. String (`java.lang.String`)

Strings in Java are immutable and internally represent a character sequence using a `char[]` containing UTF-16 code units.

Method (in class `java.lang.String`)
`public String(String original)`
`public String(StringBuffer buffer)`
`public String(StringBuilder builder)`
`public String substring(int beginIndex, int endIndex)`
`public String substring(int beginIndex)`
`public String concat(String str)`
`public String replace(char oldChar, char newChar)`
`public String toLowerCase(Locale locale)`
`public String toUpperCase(Locale locale)`

Table 4.2.: Augmented methods in `java.lang.String`

Table 4.2 lists the methods that have been equipped with code for propagating taint information. This seems like a short enumeration compared to the amount of methods defined in `String`, but most of the touched ones are primitives frequently called inside the others.

`replaceFirst()`, `replaceAll()`, `replace(String, String)`, `split()` and `replace(CharSequence, CharSequence)` operate on regular expressions and therefore on the functionally contained in `java.util.regex.Matcher`. With `format()` it is similar, it borrows from `java.util.Format`.

As described in section 3.5.1, `replace(char, char)` is an implicit taint source assigning inserted characters the predefined source `TaintSource.TS_CHAR_UNKNOWN_ORIGIN`).

Two interesting methods are `toLowerCase()` and `toUpperCase()`. These methods are far from trivial, because there are several locale-related special cases, as brought up in section 3.5.2 already. A corresponding example is the uppercasing from “ß” to

"SS". Another example is the Lithuanian glyph "Ì" represented by (\u00CC), which is tripling its size when getting lowercased to \u0069\u0307\u0300) (cf. [61]). In this example one code unit became three, resulting in a string with a different size. The transformation algorithm has been augmented in order to make sure additional `char`s inserted are part of the same taint range as the initial one was. Furthermore, the subsequent ranges are adjusted. Otherwise, a potential attacker could insert, e.g., such a Lithuanian glyph which would extend and lead to a misalignment between taint ranges and `char`s of the string.

4.3.3. `StringBuilder` & `StringBuffer`

Method
`java.lang.AbstractStringBuilder`
`    public void setLength(int newLength)`
`    public void setCharAt(int index, char ch)`
`    public ASB append(String str)`
`    public ASB append(StringBuffer sb)`
`    public ASB append(ASB sb)`
`    public ASB append(CharSequence s, int start, int end)`
`    public ASB delete(int start, int end)`
`    public ASB deleteCharAt(int index)`
`    public ASB replace(int start, int end, String str)`
`    public ASB substring(int start, int end)`
`    public ASB insert(int index, char[] str, int offset, int len)`
`    public ASB insert(int offset, String str)`
`    public ASB insert(int dstOffset, CharSequence s, int s, int e)`
`    public ASB insert(int offset, char c)`
`    public ASB reverse(String str)`
`    private ASB reverseAllValidSurrogatePairs(String str)`
`java.lang.StringBuilder`
`    public String toString()`
`java.lang.StringBuffer`
`    public String toString()`

`AbstractStringBuilder` has been shortened to `ASB` for better readability.

Table 4.3.: Augmented methods in `java.lang.AbstractStringBuilder` and related classes

Augmented methods in `java.lang.AbstractStringBuilder` and its child classes.

`java.lang.StringBuilder` and `java.lang.StringBuffer` both extend `java.lang.AbstractStringBuilder`. As the latter has package-private visibility, it is guaranteed that there is no other class inheriting from it.

The actual functionality of `StringBuffer` and `StringBuilder` is contained in their abstract parent class. They override all calls inherited which return an `AbstractStringBuilder` instance in order to set a more specific – in this case the exact – type (*covariance*). Additionally, methods are added for receiving more specific types (*contravariance*, in Java this is realized via overloading). `StringBuilder` directly delegates to its super class's definitions, `StringBuffer` additionally adds a `synchronized` modifier to achieve its thread safety – which is basically the only difference between these two. The only changes in these two source files had to be performed in `toString()` as this is not centrally handled in `AbstractStringBuilder`.

An example for the afore mentioned usage of context information used in order to write "situation-optimized" augmentation code is `reverse()`. An automatic instrumentation on a bytecode level would probably track every single `char` by adding corresponding tracking instructions wherever this `char` gets modified/copied. In case of the very basic operation of reversing a string, swapping a character would lead to swapping its taint information. Naive augmentation would do the same – take the last char, put it in front and create an according taint range. But as the augmentation is done by hand, one can use the knowledge on what `reverse()` does and therefore simple reverse the list of taint ranges – which is much more efficient. Admittedly, it still needs all ranges to be adjusted.

```
1  public AbstractStringBuilder append(String str) {
2      if (str == null)
3          return appendNull();
4      int len = str.length();
5
6      // taint>
7      if (len == 0) {
8          return this;
9      }
10     // <taint
11
12     ensureCapacityInternal(count + len);
13     str.getChars(0, len, value, count);
14
15     // taint>
16     if (str.taint != null) {
17         TaintInformation tI = this.taint();
18
19         List<TaintRange> r = str.taint.getAllRanges();
20
21         TaintInformation.adjustRanges(r, 0, len, -this.count);
22
23         this.taint.appendRanges(r);
24     }
25     // <taint
26
27     count += len;
28
29     return this;
30 }
```

Listing 4.1.: A simple example for augmentation in `java.lang.AbstractStringBuilder`. The code added is framed by *//taint>* and *//<taint*.

Listing 4.1 is a typical example for the modifications done to the standard library classes. It adds a check whether any further handling is actually necessary (lines 7 and 16). In the second block added, a check is done whether the to be appended `String` is actually tainted. If so, line 17 fetches the `TaintInformation` instance, or initializes it in case the `AbstractStringBuilder` instance was not tainted so far. Line 21 right-shifts all ranges bound to `str` by the current length of the instance. The newly created taint ranges are then appended to the instance's taint information container. This is a trivial example, but the principle shown in here is, more or less, the same as with all augmentations.

`AbstractStringBuilder` contains some implicit sources (like `replace(char old, char new)`, but not all methods qualifying as implicit source (by adding characters from unknown origin) are augmented to act as such – as a result of (naive) weighing the probability to find suspicious behavior against the caused noise. Setting `append(char[] str)`, for example, as an implicit source leads to a lot of noise as many JRE internals make use of such functionality. Therefore, an in-depth analysis should be done in order to decide how to handle these cases reasonably.

4.3.4. Regular expressions

Regular expressions are used to find occurrences of text matching a given pattern. They are an integral part of Java's standard library as, e.g., `String` uses them. As their implementation needs to provide high performance, they were assumed to operate directly on `char`s, e.g., by using `String#charAt()` – which would make augmentation necessary in order to keep taint for matched substrings. But it turned, usage of such functionality was very rare.

Beside the explicit flow aspect of extracting tainted characters as part of a substring, there is also an implicit one: a tainted string used as pattern influences what is matched in the end. But as Juturna does not cover implicit flows in general this is not of further interest.

The regular expression functionality contained in the Java JRE is located in package `java.lang.util.regex` and is accessed via instances of `Pattern`. They are created by a static builder function for a given regular expression – which gets parsed, normalized and compiled into an automaton.

A `Pattern` instance is then used to create a `Matcher` which is bound to the compiled pattern and a `CharSequence`[11]. The matcher provides functionality to actually find matching regions in a given `CharSequence` instance and to extract them.

In a nutshell, all methods in `Matcher` used for replacing or matching work by the same principle: The method `find()` gets called and sets the start and end indices of the next match found. Then, these indices are used to extract a substring or to replace the region spanned up by them – depending on the operation to be performed. Extracting a substring is done by using `subSequence()` on the set `CharSequence`. In case of the three string-like types discussed[12], this method simply delegates to `substring()` and therefore is taint-aware already, not requiring any further additions.

When it comes to replacing, `Matcher` offers extended functionality to use regions of the original input matched by capture groups as part of the replacement. Therefore, the replacement string is processed character-wise using `charAt()`. At this point, augmentation had to be performed. This critical functionality is located in `appendReplacement()`, which the various replacement operations internally rely on.

After understanding these processes, the actual augmentation was straightforward.

`Pattern` provides two more (static) methods of interest regarding taint tracking: `split()` and `quoteReplacement()`. The first is just a convenience function splitting up a given `CharSequence` around matches of the respective pattern using a `Matcher` and other already taint-aware functionality inside. Therefore, it needs no adjustments.

The latter is used to escape meta characters in a `String` in order to get a literal

[11] All of the three string-like types implement this interface, we will have a in-detail look at it in section 4.4.3

[12] For other implementations, this is slightly different (see section 4.4.3).

pattern. This is done by char-wise iterating over the input, copying all `char`s but escape characters. Augmentation had to be added therefor.

It might be possible, that taint information attached to a string used as pattern is not synchronized during normalization. But as the normalized pattern cannot be retrieved from outside the respective instance of `Pattern`, only as part of an error messages, this has been ignored. Therefore, there is no functionality in `Pattern` needing augmentation, only `Matcher` has been adjusted.

Chin and Wagner [40], the only other source-level augmentation approach the author is aware of, did not augment these classes, although, they mention that it should be done.

4.4. Bytecode instrumentation

Java bytecode is the set of instructions understood by a Java Virtual Machine (JVM), which may directly interpret it or compile it into machine code. Bytecode is contained in classfiles, a platform independent, binary format bundling code, constant pools, debug information, information on inheritance and further artifacts of a class, etc. [54, chapter 4].

Java bytecode is not strictly bound to Java as a programming language, although they obviously share concepts like object-orientation[13]: Beside *javac* for Java there are many other compilers for other languages translating code written in these into bytecode for them JVM, e.g., Kotlin code gets compile to Java bytecode.

Parsing and processing classfiles and bytecode is a quite common task in the Java world as there are many frameworks, testing and debugging tools utilizing the associated possibilities. Because of this, various bytecode processing libraries exist, absolving the developer from the task of having to make modifications to the bytecode directly. Instead, they offer him to work on a more abstract level, at least partially freeing him from thinking about *opcodes*, the operand stack or constant pools. They mostly differ regarding performance and ease of use. The most popular ones seem to be *ASM*[14], *BCEL*[15], *Byte Buddy*[16] and *Javassist*[17]. Explaining in detail why the last one was chosen would be unrewarding. In short, its functionality, popularity, maturity and affiliation with the renown company *Red Hat* were decisive – although ASM, which works on a lower level, seemed to offer a better performance in benchmarks. In retrospect, Byte Buddy seems like it would have been a better choice as it delivers more functionality out of the box that needed to be implemented with Javassist. Furthermore, there are some snares and bugs in Javassist that showed up during development and circumventing them

[13] But, for example, the JVM is capable of overloading return types – something the semantics of the Java language do not support.

[14] `http://asm.ow2.org/`

[15] `https://commons.apache.org/proper/commons-bcel/`

[16] `http://bytebuddy.net/`

[17] `http://jboss-javassist.github.io/javassist/`, used in version 3.21.0-GA.

was bothersome and time-consuming.

As presented in the concept chapter, Juturna needs to do bytecode instrumentation in order to mark methods as sources/sanitization functions or sinks in an arbitrary, given application. Including a component for modifying the bytecode is the way to go as access to source code cannot be assumed and manual augmentation of it would be pointless.

This instrumentation step can basically happen at two different points in time: as a preprocessing step done once, or as on-the-fly instrumentation performed during the process of "classloading" performed by the JVM. Juturna shall be portable and pluggable, therefore the latter strategy has been chosen – also providing support for bytecode not read from disk, e.g., retrieved by a network classloader like `java.net.URLClassLoader`.

As already touched before, see section 3.4.2, the best way to implement on-the-fly instrumentation is by using functionality offered by decent JVMs known as the Java agent interface, contained in package `java.lang.instrument`. Therefore, an application needs to be started with the additional `-javaagent:<agent-jar>` parameter. The JVM then calls the `premain()` method on the class defined in the manifest of the stated jar file, analogous to the well known `main()` method. This `premain()` method is given an implementation of the `Instrumentation` interface provided by the JRE. At this instance custom implementations of `ClassFileTransformer` can be registered. By doing so, these get hooked into the classloading mechanism. During the further execution of the JVM, every classfile is given to these transformers as a buffer of `byte`s, allowing them to modify the bytecode before getting processed by the JVM in the more narrow sense.

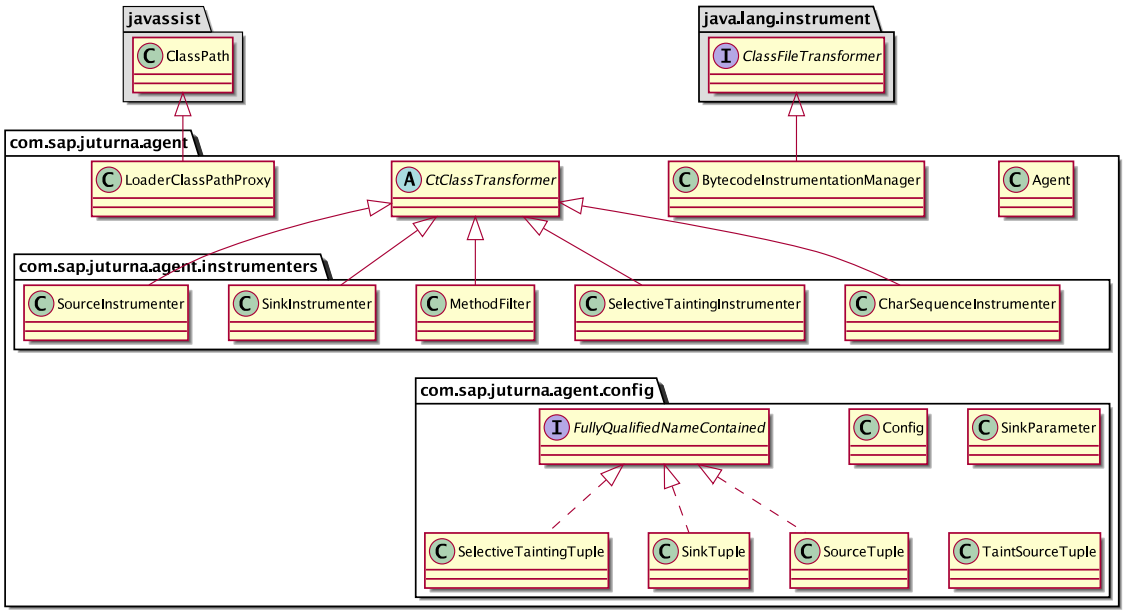

Figure 4.4.: Diagram showing the classes contained in the *com.sap.juturna.agent* package. Inheritance and implementation relations are shown, associations and aggregations are not displayed.

All code related to the bytecode instrumentation performed by Juturna is contained in `com.sap.juturna.agent.*`. The `premain()` method is placed in the class `Agent` and is responsible for registering an instance of `BytecodeInstrumtationManager` as transformer.

Additionally, it takes care of classes having already been loaded during bootstrapping of the JVM and therefore before the initialization of the agent. `Instrumentation` offers functionality to retrieve all loaded classes and to retransform them by reloading the classfiles and having them processed by the registered transformers. According to the documentation, not every JVM needs to support retransformation of classes at runtime – still, this is a widely used feature and a standard Oracle Java 8 JVM provides this functionality.

The path of the configuration file defining, among others, which methods to instrument as sinks and sources is given as a parameter to the Java agent (see figure 3.2). Therefore it is received by the `premain()` method, processing it and providing it to the various transformers. It is expected to point to a file in the *JavaScript Object Notation* (JSON) format, which gets converted into Java objects using the popular *Gson* library from *Google*. Juturna supports inheritance between configurations in order to simply derive a custom configuration from a base configuration which might, e.g., declare sinks and sources for the Java Servlet API. Examples of such configurations are shown in the evaluation chapter.

As the name implies, the `BytecodeInstrumtationManager` does not instrument itself. It is a container providing commons checks, preprocessing, logging, helper functions and dumping of modified classes for debugging purposes to the respective transformer. These transformer, implementations of `CtClassTransformer`[18], will be addressed subsequently.

To avoid cluttering the actual "instrumenters" with code for deciding which method to adjust, this functionality has been extracted into `MethodFilter`. The instrumenters use it internally by providing it with the list of methods to adjust, e.g., the instrumenter responsible for marking as sinks sets the list of declared sources, and a callback to trigger in case a suitable method has been found. Deciding, whether a method shall be instrumented or not (in reasonable time) is not as easy as one might think as we will see in the following. For the identification of methods and their declaration in the configuration, Juturna sticks to the naming scheme used by Javassist and Java Reflection (example: `java.lang.String.substring(int)`) instead of the JVM internal notation[19] (example: `java/lang/String/substring(I);Ljava/lang/String;`) as it is easier to read and write. In the following, this identifier is called the FQN of a method.

Using the FQN, one can extract the name of the classes containing sources, etc., in a preceding step and avoid parsing the bytecode of other classes by a simple lookup. Starting to list methods emitting potentially attacker controlled strings specified in the Java Servlet, one gets to the limits of this approach quite fast. `getQueryString()` declared in `javax.servlet.http.HttpServletRequest` is such a method, that should be marked as source. But as `HttpServletRequest` is just an interface and not an actual definition, one actually wants to instrument all implementations of it instead. Solely using a FQN is not sufficient as it contains the class where the method is defined, not where it has been initially declared – and this one would be needed in order to find other (re)definitions. Also, it would not be clear whether overriding methods should be instrumented or just the one actually stated. Therefore, in Juturna a user can simply state a FQN in the configuration for exactly one implementation, or prefix it with "I:" in order to have Juturna instrument all implementations and overriding redefinitions. To be able to check whether a class-to-load implements or inherits from a supertype, the bytecode of this class needs to be processed by Javassist. This basically means that all loaded classes need to be parsed. Depending on the amount of classes loaded, this can cause the startup time to be increased by a few seconds which should be considered acceptable. By adding an upstream step preparing lookup structures like `HashMap` and `HashSet`, which are then used for decision making, it is tried to keep the time consumed as low as possible.

[18] Javassist uses the name "CtClass" for "compiletime class" in order to point out the difference to the type `Class` provided by Java Reflection and representing runtime classes.

[19] [54, chapter 4.2]

Differing from the JVM internal identifier, the FQN does not contain the return type. In Java, the return type is not needed to uniquely identify a method, but on bytecode-level it is – in the first case it is not part of the method's signature, in the second one it is. With Java 1.5, among others, covariant return types (subtypes can declare a more precise type) and generics have been introduced. In order to keep the generated bytecode compatible with code compiled earlier, so called *synthetic bridge methods* are inserted, which simply delegate to the actual method[20]. As a result of adding these synthetic methods, there are multiple methods with the same FQN – simply instrumenting the bridges too would be unnecessary and therefore should be avoided. After realizing where these supposed duplicates are coming from, filtering them was quite easy as they carry a special flag.

After pointing out the framework and some snares that occurred during implementation in this section, we will look at the actual instrumentations done in the following subsections.

[20]Explaining the underlying problem would lead to far away, please refer to appropriate literature like [62, chapter 10.6.4]

4.4.1. Taint sources and sanitization functions

The class responsible for instrumenting sources and sanitization functions is `SourceInstrumenter`. It is embedded into the framework described above and gets registered at the `BytecodeInstrumtationManager` instance.

As its source code is quite concise, it is shown in listing 4.2 in order to illustrate the process of instrumentation.

```
1  class SourceInstrumenter(sources: List<SourceTuple>) : CtClassTransformer() {
2      /* ... */
3
4      private fun transformation(method: CtMethod, sourceTuple: SourceTuple): Boolean {
5          if (method.returnType.subtypeOf(TAINT_AWARE_CTCLASS).not()) {
6              throw Throwable("""${method.longName}"'s return type ("$
                   {method.returnType.name}") does not implement the TaintAware interface!
                   Therefore it cannot be marked as source!""")
7          }
8
9          val stringPoolingProtection = if (method.returnType.name == "java.lang.String") {
10             """
11                 $RETURN_VALUE_REFERENCE = new java.lang.String($RETURN_VALUE_REFERENCE);
                   // $RETURN_VALUE_REFERENCE is a placeholder for $_. Needed because a
                   literal $ cannot be used in multi-line strings.
12             """
13         } else {
14             "// just needed for instances of type String"
15         }
16
17         method.insertAfter("""
18             if ($RETURN_VALUE_REFERENCE == null) {
19                 return null;
20             }
21             $stringPoolingProtection
22             com.sap.juturna.taintHelper.THelper.get($RETURN_VALUE_REFERENCE).addRange(0,
               $RETURN_VALUE_REFERENCE.length(),
               com.sap.juturna.taintStorage.TaintSource.getOrCreateInstance("$
               {sourceTuple.taintSource}"));
23
24             return $RETURN_VALUE_REFERENCE;
25         """)
26
27         return true
28     }
29
30     /* ... */
31 }
```

Listing 4.2.: This shows the code of `SourceInstrumenter`. It is a concise example for how instrumentation is realized in Juturna and how comfortable and readable this becomes by using Javassist's functionality. It has been shortened and comments have been removed.

This instrumentation can only be applied to methods returning a type implementing the newly introduced `TaintAware` interface, otherwise a runtime exception

would be thrown when trying to access the return value's `taint` field. This gets checked at line 5.

Another pitfall, taken care of in lines 9-15, is caused by the aforementioned pooling of `String` literals. In the rare case of a method returning a pooled `String` instance, tainting this one would in fact taint the global literal. In order to resolve this edge case – that especially occurred during writing tests – a new `String` instance with a new and independent instance of `TaintInformation` is created. The chars contained are not copied by this operation as `String` is immutable and the new instance simply references the `char[]` of the original one.

As there is no way of checking whether a `String` is in the pool or not, we simply have to pessimistically assume it to be contained and therefore add this guard clause.

Starting from line 17, a simple call gets inserted, using `THelper` to get the `TaintInformation` instance of the taint-aware type and to add a range spanning the complete string setting the taint source category configured as origin. Finally the value gets returned.

Javassist made implementing this fairly easy, as `insertAfter()` takes care of compiling the given Java code to bytecode and inserting it on all possible return paths.

Although being distinguished on the semantic level, sources and sanitization functions are the same on configuration and implementation level. The different level of the assigned taint source category is used to decide whether a taint range is positively or negatively tainted.

4.4.2. Taint sinks

In order to add code to a sink method, the index of the parameter to be checked needs to be given[21]. Before inserting corresponding code, this time in front of the existing instructions, a check is done whether the type of the parameter is taint-aware and therefore actually capable of being tainted.

The added code simply delegates the task of taint checking to the `com.sap.juturna.taintCheck` component by calling `TaintCheck.checkSink()` and providing it with the options set for this sink in the configuration. According code is added for every parameter that needs to be checked.

Juturna's taint checking capabilities are quite extensive and will be outlined subsequently. An example for a corresponding configuration is given in listing 4.3.

[21] Currently Juturna does not support simply checking all parameters of a method regarding their taint state. Especially with variadic parameters this is a limitation. But as this problem can be resolved using `THelper`, it can be considered acceptable – especially for a prototype.

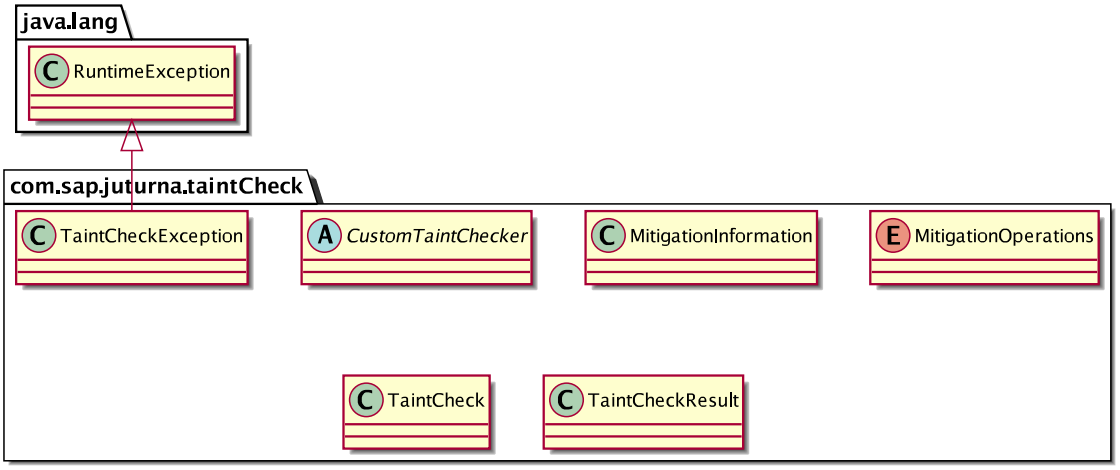

Figure 4.5.: The diagram shows the classes contained in the `com.sap.juturna.taintCheck` package.

```
{
  "sinks": [{
    "fqn": "java.lang.String.replaceAll(java.lang.String, java.lang.String)",
    "parameters": [{
      "paramIndex": 2,
      "forbiddenSources": [
        "*", "!ASOURCE_THAT_IS_OK"
      ],
      "mitigation": "CUSTOM:com.example.MyCustomTaintCheck"
    }]
  }],
  "taintSourceCategories": [{
    "name": "A_SOURCE_THAT_IS_OK",
    "severity": "POTENTIAL_LAUNDRY"
  }]
}
```

Listing 4.3.: The listings presents an example of a configuration using advanced options for sink instrumentation. `replaceAll()` gets instructed to raise a taint error in case the second parameter, containing the replacement, is (at least partially) originating from any source but "A_SOURCE_THAT_IS_OK". The chosen mitigation strategy is to delegate handling to a provided implementation of `CustomTaintChecker`. The first parameter is not considered.

For every parameter of a method, one can define a list of forbidden taint source categories and a mitigation strategy. The function deciding whether a "taint error", as Haldar et al. [4] called it, needs to be raised is given the set of taint source categories bound to the string to be checked and the list of forbidden ones. The list of not-allowed origins can contain the wildcard "*" forbidding all taint source

categories, or, in case more control is needed, names of defined taint sources. Taint sources having the level `SANITIZATION_FUNCTION` are not included in the wildcard – if desired, they need to be listed explicitly. Additionally, exclusions from the wildcard can be made by prefixing the name of a taint source with "!".

In case a taint error gets detected, the configured mitigation strategy gets applied. There are four strategies defined in `MitigationOperations` enum:

- **`LOG`:** Simply logs the incident. This does not prevent the tainted value to flow into the sink and is considered to be helpful for developers searching vulnerabilities in their own code base.
- **`THROW`:** Causes a `TaintCheckException` to be thrown preventing the, potentially malicious, string from actually entering the sink method. Throwing an exception is the appropriate way of preventing the vulnerability to be exploited as it does not alter the semantics nor harm the integrity of the instrumented method in a way, e.g., an early return would do. Still, it should not be thoughtlessly as it interrupts the application and might lead to unexpected, unwanted behavior.
- **`PASS`:** This "strategy" actually does nothing, neither writing a log entry nor preventing the string from flowing into the sink.
- **`CUSTOM`:** The most powerful mitigation strategy. It can be used to invoke provided code taking care of the situation[22].

  As shown in listing 4.3, the FQN of the class needs to be stated and an according class inheriting from the abstract `CustomTaintChecker` class has to be available in Java's classpath. It is then loaded and invoked at runtime using Java Reflection.

  The checker is given the tainted value, the list of forbidden sources and the result of the internal check already performed. It returns an instance of `TaintCheckResult`, indicating whether a taint error shall be raised and a log entry needs to be created. Additionally, the custom taint checker can override the parameter in question[23].

  Using this mechanism, handlers could be added performing a taint source specific, automatic escaping etc..

[22] Accordingly, one should exercise caution when using this feature. Further measures should be taken in order to avoid abuse of this functionality, e.g., by using an adjusted `java.lang.SecurityManager` to restrict the possibilities of the custom implementation.

[23] This is experimental as the parameter might be declared final in Java and this information is not preserved when compiling into bytecode. Therefore this could result in a "lost update" as the subsequent instructions not necessarily need to use the same reference. This needs further investigations.

4.4.3. Handling the `CharSequence` interface

`java.lang.CharSequence` is an interface describing implementations behaving like a readonly source of characters. Among others, it is implemented by `String` and `AbstractStringBuilder`. It belongs to the core classes of the JRE, putting it in the focus.

But augmentation can only be done for its manifestations, not for the interface itself – therefore, to fully cover them, project specific implementations of `CharSequence` may need to be augmented manually. Beside the ones mentioned above, the author is not aware of popular `CharSequence` implementations being used as – or at least on the same level as – strings.

The idea of implicit sources, i.e., marking characters without history when becoming part of a string, has been mentioned multiple times by now. `String#replace(char old, char new)` has been used as example for them. A `CharSequence` instance is, as the name implies, just a provider of n `char`s and therefore should be treated in a similar way by methods like `String#replace(CharSequence old, CharSequence new)`.

Instead of adding the same additional code to all methods operating on `String`, `StringBuilder` and `StringBuffer`, this has been realized using another bytecode instrumenter (`CharSequenceInstrumenter`). Avoiding some extra changes to the standard classes probably would not be worth the effort, but, as we will see further down, the instrumentation is able to cover an additional case in which taint information could be lost.

`CharSequence` declares two methods of interest: an override of `toString()` and `subSequence()` which returns a `CharSequence`. In all methods in the three string-like types which handle `CharSequence` instances, their `toString()` methods get called – therefore, it needs instrumentation.

The instrumentation is only performed for classes implementing `CharSequence` but not `TaintAware`: subtypes of the latter already supply the returned `String` with taint information when available and this should not be overwritten.

The code added to the end of `toString()` then simply creates a new `String` instance[24], initializes its `TaintInformation` instances and adds a range covering the whole string setting the predefined `TaintSource.TS_CS_UNKNOWN_ORIGIN` as source category.

With `subSequence()` it is basically the same: a not-taint-aware implementation could return a potentially taint-aware type but without information attached. If this is the case, the appended code marks the origin.

Problems occur in case `toString()` and/or `subSequence()` are inherited (from a class that does not implement `CharSequence` itself). Then, no definitions exist in the according classfile. Modifying the supertype would be dangerous, therefore redefinitions of the methods missing are inserted in an upstream step before the code described above gets appended to them[25].

[24] This is done because of string pooling, see section 4.4.1

[25] In case of classes already loaded into the JVM which are inserted into the “instrumentation

4.5. Selective tainting: integrating static analysis

In section 3.6.2, the idea of tracking taint selectively, switching between unaugmented/taint-unaware and augmented/taint-aware code, was introduced and outlined. The advantages of the approach presented in this work have been highlighted. In this section we will have a look at the prototypical implementation, before we finally see it in action and discuss the results of performed benchmarks in section 5.3.

The implementation consists of the *JOANA-Adapter*, another bytecode instrumenter contained in the Java-Agent component and further modifications to the standard classes.

Before we will dive into the details, the sequence of actions performed will be presented from a user's blackbox perspective. This shall help getting a better overview.

The JOANA-Adapter receives, besides the classfile containing the entry point, e.g., the `main()` method of an application, a simple file denoting the names of methods declared as sources and sinks. It then computes which call instructions in the bytecode are only operating on instances of string-like types that are guaranteed to not originate from a source, nor being influenced by one doing so, or which are simply not flowing into a sink. The comprehensive definition of a method call considered safe has been given in section 3.6.2 already. Those instructions can be considered "not security-sensitive parts". The terms "safe calls" and "not security-sensitive instances"/"safe instances" will be used in the following in order to refer to them.

This list of safe call instructions then gets exported as JSON, ready to be put into the main configuration of Juturna. Running an application in an accordingly configured JVM triggers the `SelectiveTaintingInstrumenter` to make the listed call instructions invoke methods containing taint-unaware code instead of the augmented, taint-aware ones. As a result, propagation of taint information is not going to be performed at these points when executing the application. This is not directly apparent to the user – but he might, in the best case, notice the improved performance caused by not performing unnecessary taint tracking operations. Figure 4.6 illustrates this setup.

Right now, only `java.lang.String` has been adjusted in order to be ready for selective taint tracking. This is because the time was limited and adding support for `java.lang.StringBuilder`, etc., works the same and is not needed for an (initial) evaluation of the concept. Adjusting these – or automatizing the process as sketched in the next subsection – could be part of followup projects taking the approach further.

pipeline" by the aforementioned retransformation are, according to the documentation, not allowed to get new methods added. Because it is known whether a class is getting retransformed or initially loaded the according step can be skipped and a warning can be presented instead.

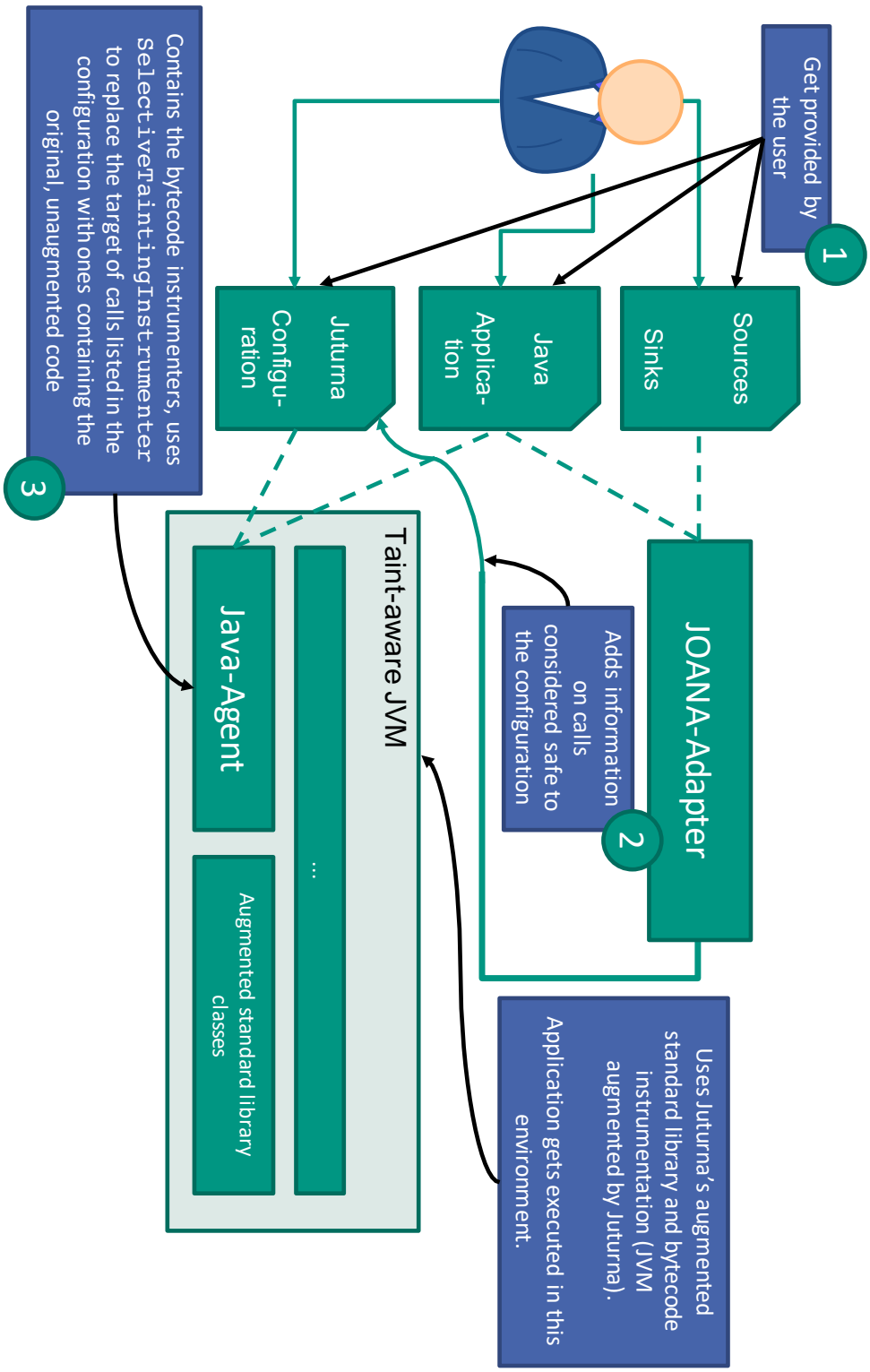

Figure 4.6.: The figure gives a schematic overview on how the components realizing selective tainting are linked together. The numbers indicate the order of the processing steps, dashed lines show dependencies.
The user icon has been taken from *openclipart.org*.

4.5.1. Providing taint-aware and taint-unaware methods

Although this was not the first task tackled, it makes sense to describe it first as switching between augmented and unaugmented code obviously requires both to be available. As a first step, the code of the original methods has been reinserted into `java.lang.String`.

Then, the names of the added methods have been prefixed with "`___`" (three underscores) to distinguish them from the augmented ones. As these methods quite often delegate tasks among themselves, calls to other reinserted methods needed to be adjusted in order to have reinserted methods only call unaugmented methods. A concise example for such a reinserted and renamed method is shown in listing 4.4.

Because of these internal linkings, not only prefixed versions of augmented methods needed to be inserted, but also methods internally using them. In case of `String` this only concerns the method `trim()`, internally calling `substring()`. Simply changing the call in the already existing definition is obviously not an option as this would affect all invocations on all instances – not only on the ones certified safe.

```
1  public String ___toUpperCase() {
2      return ___toUpperCase(Locale.getDefault());
3  }
```

Listing 4.4.: Example of a method being reinserted, renamed and adjusted in order to work as an unaugmented replacement in the selective taint tracking mechanism.

After having seen which adjustments have been done to the standard library, some key aspects can be pointed out more clearly in order to make sure the idea is understood: The selective approach decides in a first step whether to call the taint-aware or the taint-unaware method. In case of the latter, the call target of the call instruction gets replaced in the second step in order to address the unaugmented method instead.

Different string instances might be assigned to a variable a call is invoked on, depending on the execution path taken. Only if it can be assured that all possible assignments are not security-sensitive such a modification of the call target is valid. If there is one exception or uncertainty, it must not be performed as conservative approximation has to be applied here in order to preserve correctness.

Methods like the unaugmented `trim()`, internally calling a "replaceable" method, can only be modified if all possible `String` instances it could be called on are not security-sensitive. This cannot be expected for library functions as there might be may calls towards it and one call on an unsafe instance would be enough for denying the replacement. Therefore, the prefixed copy gets added – moving the chance to replace the call up the "tree", like shown in figure 4.7.

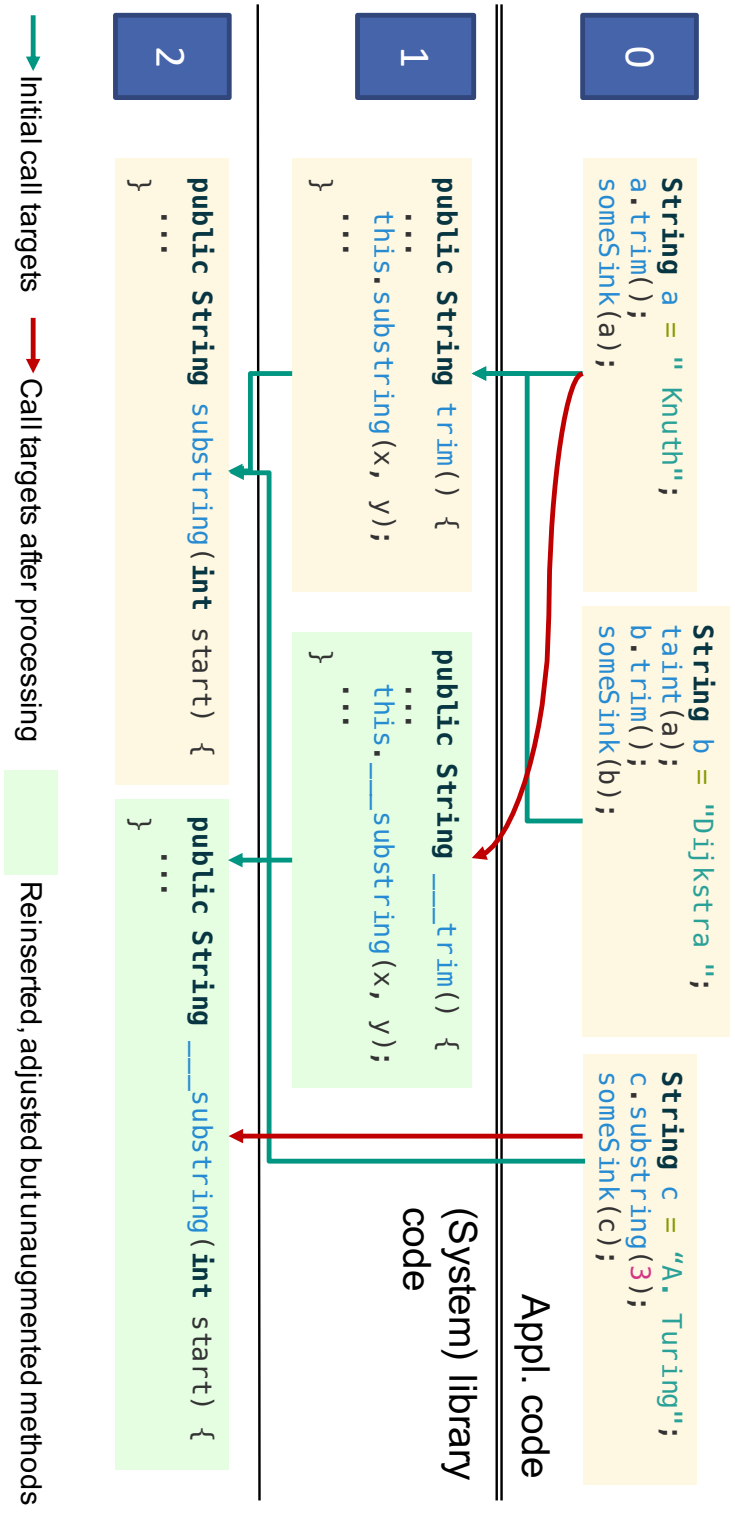

Figure 4.7.: This example tries to demonstrate the importance of avoiding calls being "funneled" to early in the "call tree" leading to non-applicability of the selective taint tracking approach.

We assume that figure 4.7 shows all calls that can possibly happen during an execution and that `someSink()` indicates that the given instance is flowing into a sink. The instance assigned to `b` has to be considered security-sensitive, the identifiers `a` and `c` are referencing not security-sensitive strings.

`this` inside of `trim()` can point to the `String` objects assigned to `a` and `b` respectively. As `b` is security-sensitive, `trim()` might operate on a tainted instance – replacing the call to `substring()` inside `trim()` with a call to `___substring()` is therefore not allowed. But as the system is able to switch the call on level 0, where there is only one instance possibly assigned to the respective identifier on which each call can be invoked, performing taint tracking on `a`'s object can still be avoided: after processing `___trim()` will be called on the instance assigned to `a`.

The approach can only be useful when switching can be done in the application code (level 0) and calls from different parts of the application with a different security sensitivity do not converge/are not summed up at the public entry points of (standard) library classes. In other words, a choice between augmented and additional, unaugmented methods must be possible at the boundary between application code and (system) library code. Just imagine, `___trim()` would not be provided in the example – the instance assigned to `a` would be handled by `substring()` because `trim()` does not only get called on instances not being security-sensitive.

In order to record which methods might be considered for replacement, i.e., augmented methods for which an unaugmented pendant exists, the Java annotation `@MightBeReplacedWithOriginalCode` has been added to them. As the annotation is retained at runtime, the `SelectiveTaintingInstrumenter` initially builds a list of those and warns whenever configuration tells it to replace something else.

Method (in class `java.lang.String`)
`public String substring(int beginIndex, int endIndex)`
`public String substring(int beginIndex)`
`public String concat(String str)`
`public String replace(char oldChar, char newChar)`
`public String toLowerCase(Locale locale)`
`public String toUpperCase(Locale locale)`
`public String trim()`

Table 4.4.: Methods in `java.lang.String` for which an unaugmented companion exists. Therefore they are ready for being replaced by the selective taint tracking approach.

Table 4.4 lists the methods that might be replaced by `SelectiveTaintingInstrumenter`. When comparing this table with table 4.2, one can notice that the latter additionally contains some constructors but misses `trim()`. Constructors are,

at the moment, not covered by the selective approach as constructors follow a fixed name scheme and therefore cannot be renamed. Further investigations need to be done in order to find a way to resolve this issue. But, as the costs of the additional functionality added to them is very low in execution, this does not matter too much anyway.

The described process of adjusting library classes could – in followup projects – be automated and, e.g., done by another bytecode instrumenter or a onetime preprocessing step. This one would take the original classfile and copy the bytecode of the original methods into the augmented classfile. Which method to copy can be determined by looking for the mentioned annotation in the augmented classfile.

4.5.2. JOANA-Adapter

In the last subsection the preparations needed for step two, switching between unaugmented and augmented code, have been explained. Additionally, the fundamental question when to replace a call has been answered in detail. With this knowledge, we will have a look at how the JOANA-Adapter detects matching call instructions.

As the name implies, the purpose of the JOANA-Adapter is to use the power of static IFC analysis mechanisms implemented in JOANA in order to provide Juturna with the information required. The JOANA-Adapter itself is not conceptionally limited to `String` which currently is the lowest common denominator in the prototypical selective taint tracking extension. The adapter is able to search for call nodes invoked on not security-sensitive instances of any type.

Subsequently, the implementation of the JOANA-Adapter will be described. As this should not require to have worked with JOANA and dived into its graph representation before reading this section, it will be explained on a more abstract level. Readers who are interested in details not provided in here are referred to the source code itself, contained in package `com.sap.juturna.joanaAdapter`.

As explained in section 2.3.2, JOANA works on a graph data structure called *System Dependence Graph*. A SDG represents an application by nodes, e.g., statements, predicates, etc., and edges showing their relations/dependencies. The different kinds of nodes relevant for the adapter will be briefly described when explaining the algorithm.

During initialization, the adapter configures the JOANA library according to its needs. Most noteworthy it disables computation of implicit dependencies, reads in sources and sinks from a given configuration, sets the specified classpath used for looking up classfiles, defines the main entry point of the application and finally lets JOANA construct the SDG.

The actual analysis is then performed in `analyze()`. Listing 4.5 shows the code, trying to hide parts whose details are not essential behind helper functions. Below the listing, the code will be explained stepwise.

The algorithm assumes the following, generic taint propagation policy: A value, in the following that might be a primitive value or an object, returned by a function marked as source is tainted and every object having at least one field/member referencing a tainted value becomes tainted itself. A source is a function returning tainted values, a sink is a function not allowed to receive tainted values via its parameters. In case of the string-like types an instance of considered tainted if it or its internal `char[]` is (partially) originating from a source or was explicitly influenced by a tainted value.

```
fun analyze(sdg: SDG, sourceFQNs: List<String>, sinkFQNs: List<String>): List<SDGNode> {
    // "data class" is shorthand in Kotlin for declaring a record-like structure
    data class CallNodeInfo(val callNode: SDGNode, val thisParameter: SDGNode, val
    thisParameterMemberNodes: LinkedHashSet<SDGNode>)

    val entryNodes: List<SDGNode> = getEntryNodes(sdg)

    val sourceEntryNodes: List<SDGNode> = filterByName(entryNodes, sourceFQNs)
    val sinkEntryNodes: List<SDGNode> = filterByName(entryNodes, sinkFQNs)

    val sourceNodes: List<SDGNode> = getAffiliatedExitNodes(sourceEntryNodes)
    val sinkNodes: List<SDGNode> = getAffiliatedFormalInNodes(sinkEntryNodes)

    val rawCallNodes: List<SDGNode> = getAllCallNodes(sdg)

    val processedCallNodes: List<CallNodeInfo> = processCallNodes(rawCallNodes)

    val slicer = SummarySlicerForward(sdg)

    var changed = false
    do {
        changed = false

        val nodesInForwardSlice: Collection<SDGNode> = slicer.slice(sourceNodes)

        for (item in processedCallNodes) {
            if (item.thisParameter in nodesInForwardSlice) {
                continue
            }

            if (isAny_thisParameterMemberNode_containedInSlice(item, nodesInForwardSlice)) {
                sourceNodes.add(item.thisParameter)
                changed = true
            }
        }
    } while (changed)

    val chopper = NonSameLevelChopper(sdg)
    val nodesInChop = chopper.chop(sourceNodes, sinkNodes)

    val benignCallNodes = ArrayList<SDGNode>()

    for (item in processedCallNodes) {
      if (item.thisParam !in nodesInChop) {
        benignCallNodes.add(getNodeOfTargetOfCall(sdg, item.callNode))
      }
    }

    return benignCallNodes
}
```

Listing 4.5.: Compressed code of JOANA-Adapter's `analyze()` function contained in `com.sap.juturna.joanaAdapter.JOANAAdapter`.

"Entry nodes" represent the entry point of a method. As mentioned before, a SDG is a collection of program dependence graphs. Therefore, the entry node of a method is the root of such an embedded PDG and all further nodes of this method

are necessarily linked to it, either directly or indirectly. They are retrieved in line 5. The helper functions called in line 7 and 8 then select the ones representing methods declared as source or sink.

The returned value of a method is represented as a dependence towards its "exit node(s)". In case of a method declared as source, these are tainted and one wants to find their dependents later on. Therefore, as a first step, the function called in line 10 fetches all related exit nodes.

A (forbidden) taint flow occurs as soon as a formal parameter ("formal-in node") of a method which is declared as sink points to a tainted value[26]. Because of this, the formal-in nodes affiliated with sink methods are fetched from the graph in line 11.

The last initialization step is to get a list of all nodes representing an invocation ("call nodes") and to preprocess them in lines 13 and 15. `preprocessCallNodes()` doest quite a lot under the hood. First, it takes the list and sifts out calls to static methods, constructors and all other calls not operating on one of the taint-aware types. Second, it creates a list of `CallNodeInfo` objects (line 3) in order to store additional information along with the call node: the `thisParameter` node representing the set of possible objects the call might be invoked on[27] and a set of nodes representing fields/members belonging to the `thisParameter`.

These "parameter member" nodes are linked to the `thisParameter` via "parameter structure" edges indicating that these are accessible via a field of the node at the other end of the edge.

Now we are getting to the core: they are of interest because such a node representing a tainted value – when being linked to an instance of a taint-aware type – signals that this instance's internals are linked to tainted values and therefore the instance itself needs to be considered tainted too according to the propagation policy stated above. In other words, if nodes connected to a node representing a string via parameter structure edges are tainted, a tainted value is contained in the string.

In line 17, a forward slicer gets initialized[28] which is used to compute all nodes depending on, at least, one node out of a given set of start nodes.

Then, the set of nodes possibly being tainted by possibly having a (transitive) data-dependency towards a source's exit node gets iteratively computed during a *fixed-point iteration* – starting from line 19.

In line 23, the forward slicing operation is performed, followed by looping over the `processedCallNodes` list: If any of the parameter member nodes is contained in the slice, the object having fields pointing to these values becomes tainted too. Therefore, it needs to be added to the set of possible sources in order to find out which further parts of the application might be influenced by, e.g., receiving this

[26] In rare cases this might approximate a little to conservative as the parameter does not necessarily reference the tainted value at the time the method in question gets invoked.

[27] The static analysis does not necessarily needs to be sure on which instance a method might be invoked as this might be determined at runtime. By having edges to multiple other nodes, it is possible for a single node to "represent a whole set".

[28] Kotlin abandoned the `new` keyword in front of constructor calls.

object as a parameter. This step of adding "auxiliary sources" is necessary because the slicing performed by JOANA does not consider the dependency represented by those parameter structure edges as they are merely an auxiliary construct and not part of the actual PDG.

In order to compute the complete slice, containing all nodes transitively dependent from any of the declared sources, this is repeated until a fixed-point is reached, i.e., `sourceNodes` does not grow any more.

Reaching this point, `sourceNodes` contains nodes representing the declared sources and the added auxiliary sources. `nodesInForwardSlice` additionally contains all nodes that depend on by a data dependency – but not all of them are security-sensitive, as the nodes representing the sinks might not depend on them. Just think of a string emitted by a source but not flowing into a sink once more, the calls invoked on this string are not security-sensitive.

The next step is to compute the "intersection"[29] of the forward slice, starting from the nodes in `sourceNode`, with the backward slice, starting from the ones in `sinkNodes`. JOANA provides a "chopper" for this task.

In the final step, all call nodes whose "this parameter" is not contained in this chop are added to the resulting list of call nodes considered not to be security-sensitive, i.e., they are safe to be replaced.

In order to have the bytecode instrumenter contained in the Java-Agent component replace those calls, a way of addressing a single call operation, i.e., a specific occurrence of an `invoke*` opcode in the JVM bytecode, had to be found. Using the FQN of the method containing the invocation opcode combined with the bytecode index[30] of the instruction turned out to be the best scheme as it is unambiguous and works smooth with JOANA on the one and Javassist on the other side.

As a post-processing for the not security-sensitive call nodes determined, they are matched against a list of methods declared as replaceable in order to keep the generated output more concise. Additionally, their bytecode index gets extracted and they get grouped by the method they are contained in. As the bytecode instrumenter uses Javassist's naming scheme for methods, further functionality has been implemented into the JOANA-Adapter to convert the FQNs from the JVM internal notation used by JOANA to the one used by Javassist.

The information provided by the JOANA-Adapter then needs to be placed in the main configuration of Juturna. This split-up into a configuration for the adapter, running as a preprocessing step, and one for Juturna's core components was mainly done to be more flexible when experimenting in different evaluation scenarios.

Additionally, the adapter, and also JOANA itself, has some restrictions as it will be pointed out in section 5.4. These prevent its usage in some scenarios and

[29] As mentioned before, see section 3.6.2, the performed operation is not actually the computation of an intersection. Instead, a chop is computed.

[30] The bytecode index is the offset of the first byte of an instruction containing its opcode. The opcode is always one byte long and is followed by the fixed amount of fixed size operands. [54, Section 2.11]

therefore tight coupling did not seem advisable. Currently, the taint tracking system of Juturna can be used completely on its own, the selective taint tracking approach is just an extension.

An example showing a small test application and the according output of the JOANA-Adapter will be presented in section 5.3.

4.5.3. Switching between unaugmented and augmented code

The final piece of the puzzle is the `SelectiveTaintingInstrumenter`. It is part of the Java-Agent component and therefore of Juturna's core. Via the configuration it retrieves a list of methods, represented by FQNs, containing at least one invocation instruction addressed via its bytecode index. These have been determined to be not security-sensitive by the JOANA-Adapter and therefore might be replaced.

The task of this instrumenter is pretty simple as it basically just checks whether the current target of the call is in the list of switchable methods and, if that is the case, changes the target by adding the `___` prefix to the called method's name. Some additional checks are performed in order to inform the user whether all replacements could be applied.

The implementation is a little more complicated due to the fact that modifying a method's bytecode with Javassist causes the library to directly apply the modifications and regenerate the bytecode of the whole method. As it turned out, the regenerated bytecode is not necessarily equal to the original one regarding the amount of instructions. This results in shifting further invocation instructions within the same method and therefore changes their bytecode indices – raising problems when trying to modify more than one call instruction per method. A workaround could be found to resolve this problem.

In case a call cannot be replaced due to an issue in the `SelectiveTaintingInstrumenter` or in any other part of the instrumentation pipeline, the concept of selectively removing taint tracking for not security-sensitive areas ("secure by default") instead of adding it to sensitive parts shows its advantages: an unnecessary overhead not being removed is for sure better than a necessary security check not being added.

4.6. Testing

Software testing is essential, especially when code becomes more complex. Testing is considered a standard for properly crafted software nowadays. Additionally, these tests serve as another kind of documentation, beside the comments in the classes themselves.

Juturna's core components consist of $\approx$ 30 Kotlin files – plus the augmented standard library classes – which are covered by $\approx$ 20 test classes based on the popular testing framework *JUnit*[31]. The tests are mostly unit tests, but also some

[31] `https://junit.org/`

small integration tests are included. In total they contain ≈ 370 assertions.

In order to assure that the augmentations done to the standard library classes do not change their expected and observable behavior every modified method has been tested regarding their predefined behavior and additionally regarding its handling of taint information. In order for this to work, the tests are executed in a taint-aware JVM augmented by Juturna.

Testing the bytecode instrumenters has been done by applying them to specially adjusted classes, followed by checking their runtime behavior. This seemed to be a better way then verifying the modifications on bytecode level.

Although Dijkstra said that "program testing can be a very effective way to show the presence of bugs, but is hopelessly inadequate for showing their absence" [63], it surely reduces the probability of their existence. Because these tests helped to track down bugs in an early state, only a few painful debugging sessions were needed making the whole implementation process more efficient.

5. Evaluation

After having extensively presented and discussed the prototypical implementation, it is time to evaluate it regarding its capabilities to detect actual vulnerabilities and regarding its performance.

First, some servlets containing actual weaknesses will be run in a taint-aware Java EE environment, demonstrating that the prototype of Juturna is already capable of protecting simple web applications by detecting the contained flaws. Then, the performance of Juturna will be measured and discussed before evaluating the approach of selective taint tracking. The chapter will conclude with an exhaustive discussion of currently open problems and possible solutions as well as limitations of the concepts and technologies employed.

All benchmarks and tests have been performed using Oracle's JRE in version 1.8.0_111, running on a MacBook Pro[1].

5.1. Detecting existing vulnerabilities

The prototypical implementation of the presented taint tracking system is not yet capable of handling full-blown Java EE web applications incorporating complex frameworks properly. Therefore, an evaluation using the popular and deliberately vulnerable *OWASP WebGoat*[2] based on the Spring framework would not have been feasible while keeping the preparation and configuration comprehensible – we will come back to the issues that arose at the end of the evaluation chapter, see section 5.4.

Instead, the evaluation has been done on a selection of examples contained in the *Standford SecuriBench Micro*[3] benchmark suite. Version 1.08 has been used. The project's objective is to provide a set of basic to complex Java EE example servlets for the evaluation of static security analyzers – nevertheless, it is also valuable for the verification of dynamic security tools. It consists of almost 100 test cases implemented as isolated servlets, most of them containing weaknesses promoting reflected XSS attacks.

Due to its popularity, Apache Tomcat 9.0.0.M26 has been chosen as the underlying Java EE implementation providing support for the web profile – but basically every compliant servlet container could have been used instead. The SecuriBench Micro benchmark has simply been deployed as web application.

[1] Mid 2015, macOS 10.13.2, 2.2 GHz Intel Core i7

[2] `https://github.com/WebGoat/WebGoat`

[3] `https://suif.stanford.edu/~livshits/work/securibench-micro/index.html`

```java
public class Aliasing5 {
  private static final String FIELD_NAME = "name";
  protected void doGet(HttpServletRequest req,
  HttpServletResponse resp) throws IOException {
    StringBuffer buf = new StringBuffer("abc");
    foo(buf, buf, resp, req);
  }

  void foo(StringBuffer buf, StringBuffer buf2,
  ServletResponse resp, ServletRequest req) throws
  IOException {
    String name = req.getParameter(FIELD_NAME);
    buf.append(name);
    PrintWriter writer = resp.getWriter();
    writer.println(buf2.toString()); /* BAD */
  }
}
```

Listing 5.1.: Code of first test case.

```java
public class Inter10 {
  private static final String FIELD_NAME = "name";
  protected void doGet(HttpServletRequest req,
  HttpServletResponse resp) throws IOException {
    String s1 = req.getParameter(FIELD_NAME);

    String s2 = foo(s1);
    String s3 = foo("abc");

    PrintWriter writer = resp.getWriter();
    writer.println(s2);          /* BAD */
    writer.println(s3);          /* OK */
  }
  private String foo(String s1) {
    return s1.toLowerCase().substring(0, s1.length()
    -1);
  }
}
```

Listing 5.2.: Code of second test case.

The only adjustment needed to be done to Tomcat was to create `setenv.sh` in Tomcat's configuration directory. It is, per convention, called when Tomcat starts and it embodies the recommended way to set additional arguments for Tomcat and the JVM. It contains a single line of *bash* code: `export CATALINA_OPTS="-Xbootclasspath/p:<JUTURNA_JAR> -javaagent:<JUTURNA_JAR>=<TOMCAT_BASE>/juturna-config.json"`. No further preprocessing, etc., is needed in order to have Tomcat run in a taint-aware environment.

Listing 5.1 and listing 5.2 show the two examples taken from SecuriBench Micro, "Aliasing5.java" and "Inter10.java". They have been chosen because they perform some operations on the entered, tainted string, respectively transfer its value into a `StringBuilder` instance.

The sources and sinks are derived from the Java Servlet Specification. As the test cases contained in SecuriBench Micro only retrieve user input via `ServletRequest.getParameter()`, the configuration (see listing 5.3) provided to Juturna is really basic. The list bound to "additionalClasspaths" may contain additions that need to be added to the internal classpath of Javassist, which needs to resolve imported type declarations. As Tomcat does not add its libraries to the global classpath, instead using custom classloaders, the path containing these needs to be announced to Javassist on this way.

```
{
  "additionalClasspaths": [
    "lib/*"
  ],
  "taintSourceCategories": [
    {
      "name": "URL_PARAMETER",
      "severity": "ACTUAL_SOURCE"
    }
  ],
  "sources": [
    {
      "fqn": "I:javax.servlet.ServletRequest.getParameter(java.lang.String)",
      "taintSource": "URL_PARAMETER"
    }
  ],
  "sinks": [
    {
      "fqn": "org.apache.catalina.connector.CoyoteWriter.print(java.lang.String)",
      "parameters": [
        {
          "paramIndex": 0,
          "forbiddenSources": [
            "*"
          ],
          "mitigation": "LOG"
        }
      ]
    }
  ]
}
```

Listing 5.3.: Configuration of Juturna used to detect the vulnerabilities contained in the two given test cases.

The obvious sink regarding the two test cases is `PrintWriter.print(String)`. An according instance of `java.io.PrintWriter` gets provided by calling `ServletResponse.getWriter()`. Without deeper knowledge about Tomcat's internals, one does not know whether this call will return an instance of `PrintWriter` itself or of a subtype (polymorphism). Therefore, the sink needs to be declared as a possibly redefined sink ("inheriting sink")[4], causing the system to treat all redefinitions/overriding methods as sinks. But doing so might lead to unwanted side-effects: Printing user controllable input, like the requested path, to the logs may lead to unnecessary taint incidents getting reported as this printing might happen via (a subclass of) `PrintWriter`. How this could be avoided in future versions of Juturna, will be discussed in section 5.4. For this test scenario the actual implementation provided by Tomcat, `org.apache.catalina.connector.CoyoteWriter.print()`, has been configured as sink.

[4]This is indicated by the "I:" in front of the FQN. See section 4.4.2.

```
1  ===========>
2  Taint incident occurred at:
3  org.apache.catalina.connector.CoyoteWriter.print
   (CoyoteWriter.java)
4  [...]
5
6  Further information:
7  Actual sources: URL_PARAMETER
8  Forbidden sources: *
9  Mitigation strategy: LOG
10 Tainted value (up to first 300 characters; not all
   of these characters are necessarily tainted): hoar
11 TaintInformation object: TaintInformation(
12   TaintRange(start=0, end=4, source=TaintSource
     (name='URL_PARAMETER', id=-32765,
     level=ACTUAL_SOURCE)))
13 <===========
```

Listing 5.4.: Emitted log messages for first test case.

```
1  ===========>
2  Taint incident occurred at:
3  org.apache.catalina.connector.CoyoteWriter.print
   (CoyoteWriter.java)
4  [...]
5
6  Further information:
7  Actual sources: URL_PARAMETER
8  Forbidden sources: *
9  Mitigation strategy: LOG
10 Tainted value (up to first 300 characters; not all
   of these characters are necessarily tainted):
   abcHoare
11 TaintInformation object: TaintInformation(
12   TaintRange(start=3, end=8, source=TaintSource
     (name='URL_PARAMETER', id=-32765,
     level=ACTUAL_SOURCE)))
13 <===========
```

Listing 5.5.: Emitted log messages for second test case.

Requesting both servlets with "Hoare" as value for the `name` parameter, both vulnerabilities have been detected. Listing 5.4 and listing 5.5 show the corresponding log messages produced by Juturna. This proves that Juturna is capable of detecting vulnerabilities and, by configuring another mitigation strategy, also preventing their exploitation. Although the test cases presented in here are straightforward, they are not trivial. It furthermore shows that Juturna can be combined with a servlet container and yet be configured to protect (basic) Java EE web applications.

5.2. Performance benchmarks

All of the taint tracking systems for Java considered as related work have undergone some performance benchmarks. Unfortunately, the corresponding papers do not contain enough information to reproduce their setups used in order to directly compare against them.

Bell and Kaiser [31] used the publicly available *DaCapo Benchmark*[5] project, but they do not specify which information has been marked as tainted for the benchmark. Also, comparing to them is neither meaningful nor fair as their system is capable of tracking taint information for primitive data types causing increased overhead. According to them, as mentioned before, they achieved an overhead of 52%.

The system of Chin and Wagner [40] has very similar characteristics to Juturna, but the benchmark they present consists of running the Java-based discussion board *JForum* in a taint-aware JVM and sending HTTP requests from different machines, measuring the requests handled per second. As they do not state details on the setup, e.g., which URLs they requested, the setup cannot be reproduced. They state an overhead of $0-15\%$ (for the whole system, not only the string operations!), depending on the payload of the request and the configuration of their system. The

[5] `http://www.dacapobench.org/`

meaningfulness is questionable as no information is given on what kind of (string) operations are actually performed, making the result not transferable.

Haldar et al. had their system, performing only string-level tracking, undergo a micro-benchmark, focussing on string operations. It "showed no noticeable difference in execution time of the benchmark between using the original and instrumented String class" [4]. They neither specify the operations performed, nor do they give any numbers enabling a comparison.

The works combining taint tracking with static analysis mechanisms (Mongiovi et al. [45], Zhao et al. [46]) do not present any evaluations regarding the runtime overhead added by their systems.

Therefore, the performance benchmarks discussed in the following will simply be run in a taint-aware JVM augmented by Juturna and an untouched JVM serving as baseline. Comparing with other systems would be hard anyway due to the varying capabilities of the systems and Juturna's concept of taint ranges, accepting an increased computational overhead in order to allocate (massively) less memory in case of huge strings flowing through an application.

5.2.1. Setup & methodology

Trying to minimize the impact of external influences and to focus on the overhead added by the presented taint tracking approach, the benchmark is set-up as micro benchmark. In order to avoid common pitfalls when running benchmarks in the JVM, e.g., *just-in-time compilation*, *dead code elimination*, etc., and to get handling of iterations and timing for free, the popular *Java Microbenchmarking Harness* (JMH)[6] tool, being part of the *OpenJDK* project, has been used. In the configuration used it follows the summarized recommendation by Georges et al. [64] on how to rigorously benchmark Java applications by sequentially running multiple JVMs, each executing multiple benchmark iterations.

The benchmarks described subsequently measure the impact the augmentation of the standard library and the closely associated management of the taint ranges cause.

The time complexity of the additional code in general depends only on the amount of taint ranges attached to a given string, the ranges' length is irrelevant. For the method used in the benchmark, the time complexity of the actual string handling code depends only on the strings' length. Therefore, these two rule the computational overhead caused by taint tracking. In order to illustrate their influence, string length (n) and amount of taint ranges (k) have been varied:

$$n \in \{250, 10000\}$$
$$k \in \{0, 1, 3, 10, 100\}$$

[6] `http://openjdk.java.net/projects/code-tools/jmh/`

String lengths of 250 and 10000 are assumed realistic, e.g., representing an URL containing some parameters or a HTML template containing placeholders as response. The amount of taint ranges per string is expected to be very low: usually 0 or 1[7], maybe 3. But in case of the the aforementioned templates more ranges, e.g., 10 or 100, might possibly be attached.

To tell the whole truth, runtime additionally depends on which parts of a string are covered by taint ranges and their relations to the area getting modified during an operation: Replacing text on a `StringBuffer` instance by an untainted string behind the last taint range causes (nearly) no additional overhead. But replacing in front or inside the first range requires updating the first range and, in case the replacement has not the same size as the area it replaces, all further taint ranges need to be adjusted too.

Listing 5.6 shows the essential parts of the benchmark performed together with the configuration of JMH. A complete version can be found in the appendix, listing A.1.

The process of initially attaching the taint information to the different `String` and `StringBuilder` instances is done by adding k taint ranges, one for each of the last k characters. This positioning of taint ranges leads to worst case scenarios for the implementation of the taint storage engine – leading to more honest, not whitewashed results. As the taint ranges are added at runtime via `THelper`, an explicit configuration of Juturna is not necessary.

[7]E.g., the URL associated with a request would have a single range attached.

```java
@Warmup(iterations = 5) @Fork(5) @BenchmarkMode(Mode.Throughput)
@Measurement(iterations = 5) @State(Scope.Benchmark)
public class JuturnaBenchmark {
  @Param({"250", "10000"})
  static int LEN;

  @Param({"0", "1", "3", "10", "100"})
  static int TAINT_RANGES;

  String str_a; StringBuilder sb_b;
  StringBuilder sb_c; StringBuilder sb_d;

  @Setup
  public void setup() { /* ... */ }

  private static StringBuilder getInitialString(int includeKTaintRanges) { /* ... */ }

  @Benchmark
  public void a() {
    str_a = str_a.substring(0, LEN / 2).concat(str_a.substring(LEN / 2));
  }

  @Benchmark
  public void b() {
    sb_b.insert(LEN / 2, "foobar");
    sb_b.delete(LEN / 2, LEN / 2 + 6);
  }

  @Benchmark
  public void c() {
    sb_c.replace(LEN / 2, LEN / 2 + 6, "foobar");
  }

  @Benchmark
  public void d() {
    sb_d.reverse().reverse();
  }
}
```

Listing 5.6.: This listing contains the essential parts of the performance benchmark, whose results will be discussed subsequently. A full version is placed in the appendix.

Four (representative) tasks/benchmarks, a to d, have been chosen:

- **a**: Getting both halves of a `String` and concatenating them again afterwards. The second half contains k taint ranges which need to be adjusted when extracting the substring, i.e., their indices need to be shifted to the left. When combining the strings, the indices need to be adjusted again.
- **b**: Inserting the untainted string "foobar" at the middle of the string hold by a `StringBuilder`. Then removing it again. As the taint ranges are bound to the last characters, inserting at the middle need them to be adjusted. The

same hold true, obviously, for the inverse operation of removing the inserted characters again.

- **c**: Replaces the six characters starting from the middle of the string represented by the `StringBuilder` with the ones of "foobar". This is an interesting scenario as the amount of replaced characters is equal to the length of the replacement. We will come back to this later.

- **d**: Reverses the content of a `StringBuilder` instance twice in a row. This also causes the taint information to be reversed.

Besides `reverse()`, these can be considered primitives, building blocks for more sophisticated string manipulations. Therefore, they seemed to be a good choice.

Micro benchmarks are usually run multiple times until the variance of the measurement undercuts a threshold. A single run should not influence the ones following. Therefore, these tasks either do not affect the length of a string and the taint information attached, or additionally contain the inverse operation restoring the old state.

Every task is run in a taint-aware JVM for every combination of the parameters regarding string length n and amount of taint ranges k. For the baseline measurements on a taint-unaware JVM k is obviously not varied.

Based on the benchmark class shown in listing 5.6, JMH generates the actual benchmark application using Java's *annotation processor* feature and bundles it with the framework's foundations and dependencies.

5.2.2. Results

JMH tries to minimize the influence non-deterministic optimizations, unpredictable *garbage collector* runs, etc., have towards the measurements. It therefore has been configured to run every configuration of the benchmarked tasks in five separately invoked JVM instances performing five warmup iterations to ensure the JVM has had enough time to apply its optimizations and just-in-time compilation, followed by five measured iterations.

The results shown in the figures below are stated in operations per seconds (ops/s). The bars indicate the average amount of achieved ops/s and the error bars represent the standard deviation computed. All measured values have been taken into account. The measured values can also be found in table A.1 and table A.2, located in the appendix.

The first bar always displays the value measured running inside an untouched JVM, the other five are run in a taint-aware JVM whereas k indicates the amount of taint ranges attached to the `String`/`StringBuffer` instance the task is performed on. They contain either 250 (greenish) or 10000 (blueish) characters.

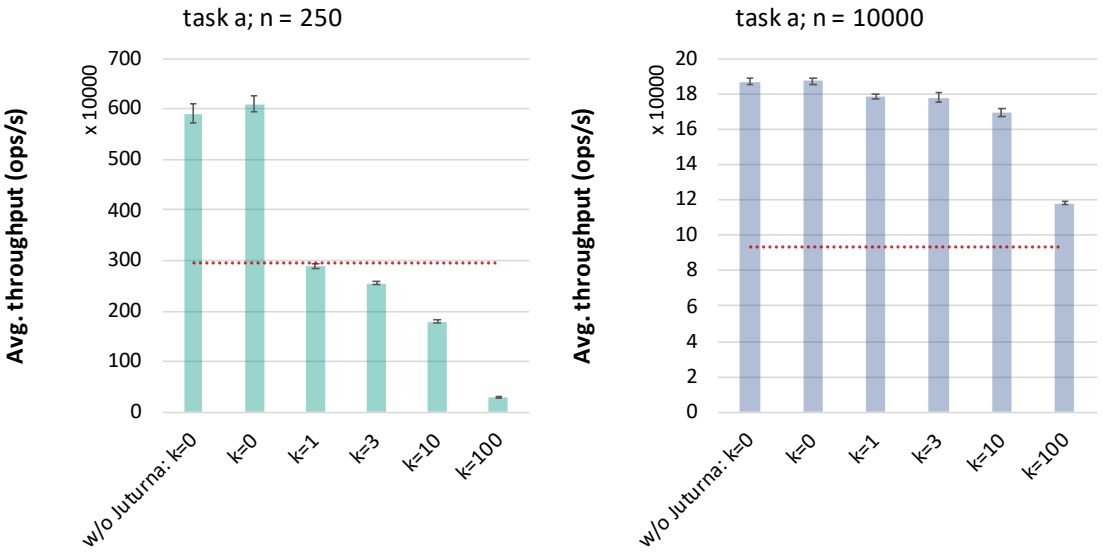

Figure 5.1.: Results for task a; $n = 250$ on the left, $n = 10^4$ on the right.

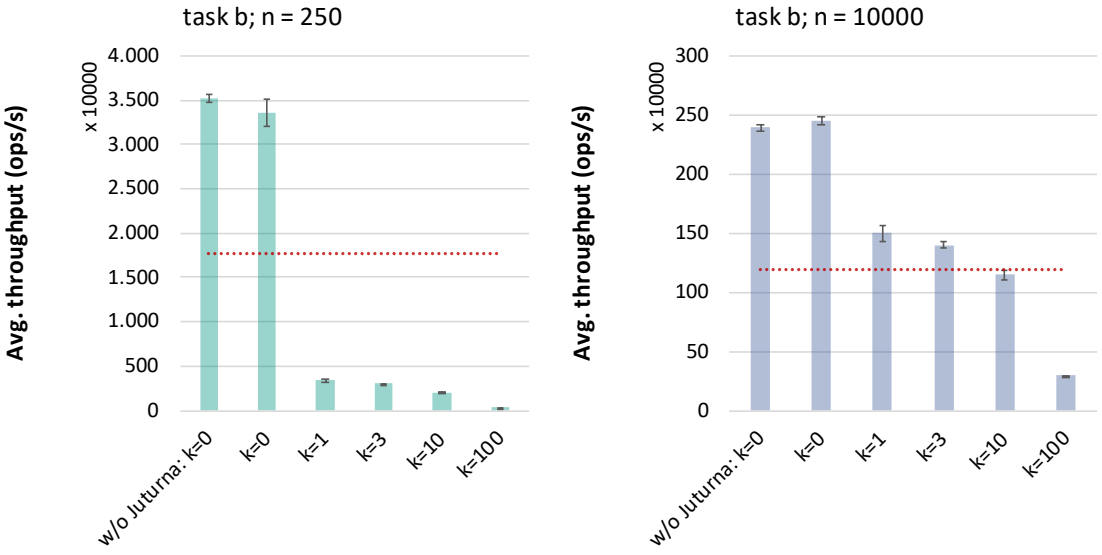

Figure 5.2.: Results for task a; $n = 250$ on the left, $n = 10^4$ on the right.

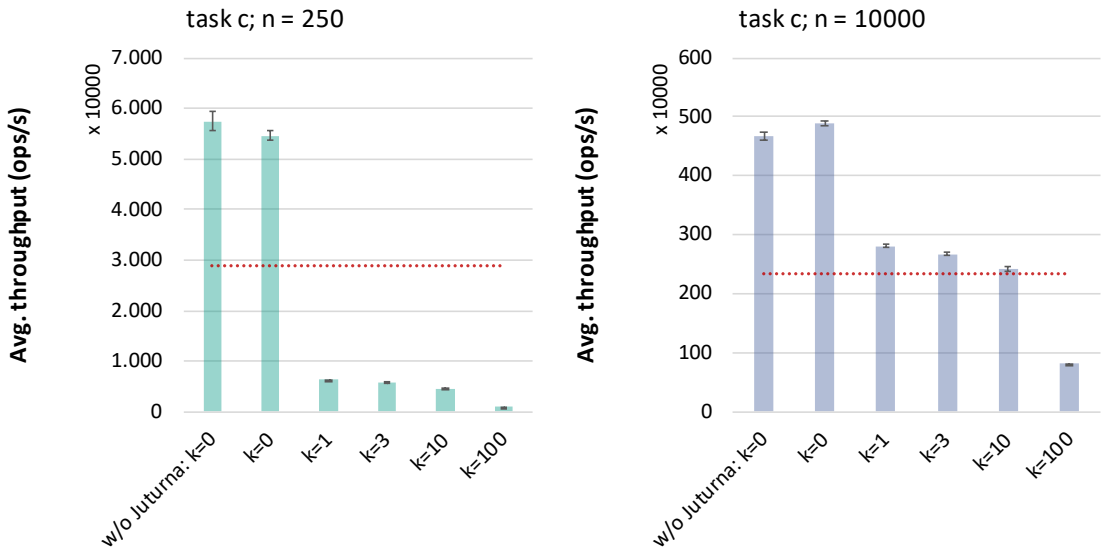

Figure 5.3.: Results for task a; $n = 250$ on the left, $n = 10^4$ on the right.

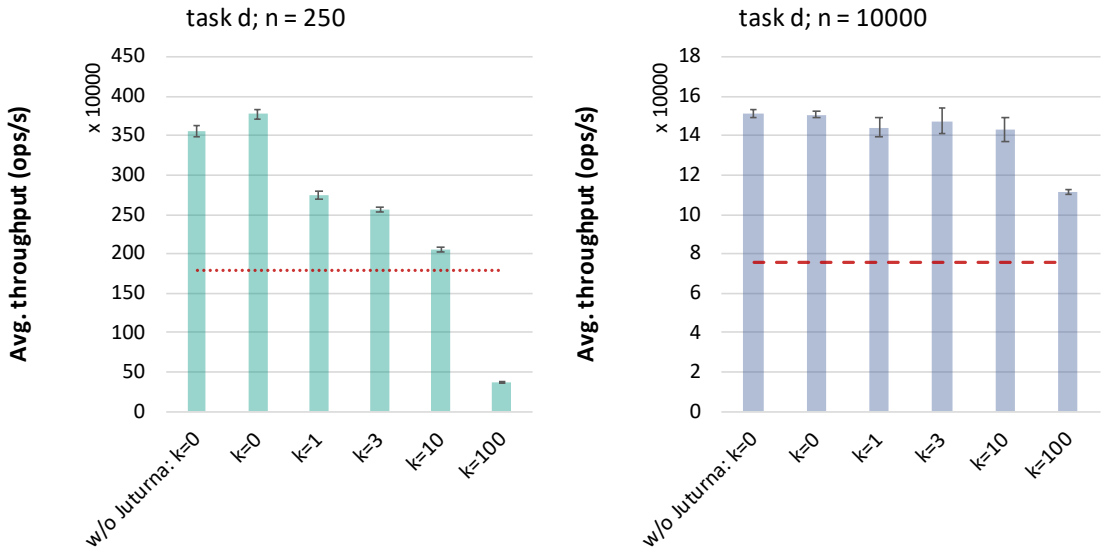

Figure 5.4.: Results for task a; $n = 250$ on the left, $n = 10^4$ on the right.

Overall, the results are no big surprise, indeed, they slightly exceed the expectations.

At first sight, the diagrams shown below might look disillusioning as the overheads added by the taint tracking system for a given configuration are constituted by the difference between its bar and the baseline bar. But one has to consider some important aspects when interpreting them: Taint tracking at runtime never comes for free, it always causes a computational overhead and an enlarged memory footprint. As a simple approximation in order to classify the results better: Even the naive approach of storing taint information in an array of type `byte/char` can be expected to approximately double the computation time of primitive operations like appending, extracting a substring, inserting and deleting for strings of a sufficient length. This can be assumed as the operations, mostly copying and allocating memory, performed on the internal `char[]` need to be performed more or less analogously on the array containing the taint information[8]. Implementations like the one of Chin and Wagner [40] are assumed to be approximated by this, achieving halve of the ops/s of the baseline configuration. This is represented by the reddish, dotted line in the diagrams.

In case of $k \in \{1, 3, 10\}$ and $n = 10^4$, performance is at least similar to the naive approximation for all tasks. Admittedly, the naive approach would be capable of handling a different taint source per character – but this is not to be expected in real use cases.

The concept of taint ranges is much more sophisticated than a simple "shadow array", e.g., it requires checks whether ranges inserted collide with existing ones, making it more costly. On the other hand, taint ranges consume much less memory than the naive approach does, as it has been shown before in section 4.2.3. This represents a trade-off regarding time and space complexity.

In most of the tasks benchmarked, the logic for manipulating the internal `char[]` is very simple and can be performed blazingly fast[9]. The computation time of the given tasks is the sum of the time needed for the actual string manipulation and the time it takes to update the taint ranges – making the actually not too slow management of the taint ranges look bad regarding its huge relative share of the total computation time needed. This relation becomes clearly visible when looking at the values measured for $n = 10^4$. The costs adjusting the taint information stays the same – with the naive approach it would increase linearly – while the time consumed by the actual string manipulation increases, therefore decreasing the overhead's relative share of the total.

Furthermore, this overhead becomes, relatively seen, much smaller when consider-

[8] For $k = 0$, the naive approach can make use the idea of Chin and Wagner to not initialize the taint array. This would eliminate the overhead for this edge case.

[9] Additionaly, due to the way the initial `StringBuilder` instances get created, their capacity is already higher than the length of the string represented. Therefore, operations enlarging the string do not cause the `chars` stored internally to be migrated to a bigger `char[]`, avoiding some copy operations that certainly would occur in a real world situation.

ing these tasks as secondary parts of an actual application which might perform some time consuming calculations, accessing files from a hard disk, querying a database or requesting some information from a web service. Coming back to the aforementioned $0 - 15\%$ overhead stated by Chin and Wagner [40], similar results should be achievable by this implementation for real world examples while keeping the memory usage low(er).

Nevertheless, suggestions to reduce the overhead measured will be presented in the open problems section, see section 5.4.2.

After putting the measurements into perspective and explaining why there is a significant overhead, some general observations will be described now.

As the pairs of measurements for $k = 0$ (with Juturna and without; for all tasks) show, the additional operations inside the standard classes' methods, checking whether taint information need to be adjusted or not, cause no overhead. Most of these checks are not actually necessary, but have been added to prevent overhead wherever possible. Starting from $k = 1$, as soon as at least one taint range exists and the aforementioned checks do not skip adjusting of taint information anymore, the overhead becomes noticeable.

Obviously, the relation between n and k is of great importance regarding the overhead caused – but this is not surprising and has already been mentioned when comparing the memory footprint of taint ranges with the naive's approach one.

The takeaway is to keep the amount of (different) taint source categories attached low in order to benefit from merging and to avoid intensive string manipulation on such strings when possible.

In all scenarios not coming below a certain ratio between n and k, taint ranges are superior to the naive approach regarding memory usage. This has been elaborated in section 4.2.3 and the gap between both approaches regarding memory usage is illustrated in figure 4.3. Measuring the memory overhead actually occurring during the benchmarks was not possible. For a configuration like $n = 10^4; k = 100$ one would assume, according to the approximations postulated in section 4.2.3, an additional memory footprint of roughly 4000 byte ($\approx$ 1000 byte per string operated on times 4) would be expected after letting Java's garbage collector remove objects not needed anymore. Therefore, the amount of memory occupied by the JVMs heap and stack have been measured using some JVM management functionality once immediately before (t_0) and after (t_1) running the benchmarks. Some tricks have been applied to ensure that the triggered, asynchronously run garbage collector had finished before retrieving the values. The differences measured between t_0 and t_1 fluctuated massively between ± 10 mebibyte, making the detection of 4000 byte impossible.

The naive approach (section 4.2.3) would be expected to occupy $4 * 10^4$ byte for this scenario – admittedly not an amount to worry about. But with applications holding a lot of huge strings, and there are plenty of them out there, the linear correlation between n and the amount of memory needed is clearly unfavorable.

In order to eliminate all the assumptions regarding plausible values for k and n made above and to get rid of the aberrations introduced by the benchmark itself, the next step would be to measure the performance overhead caused by applying Juturna to real Java EE applications, containing all of the much more time consuming tasks mentioned above. But currently, this is not yet feasible as pointed out in the foregoing section already.

5.3. Combining dynamic and static analysis

After having shown some benchmarks regarding the prototypical implementation's performance and having discussed the results in the last section, the focus of the last evaluation is to provide a simple example demonstrating how JOANA-Adapter works and how it can be used to reduce the overhead taint tracking causes by a selective augmentation approach.

5.3.1. Setup & methodology

The aforementioned performance benchmark is based on JMH – which adds quite some complexity to the build process, as it generates a lot of code for the benchmark to be ready. Still, JOANA-Adapter should be able to handle it. But as the current evaluation is not about measuring performance with maximum precision, it seemed advisable to present a more concise and understandable example not spanning over multiple files of source code.

Therefore, a minimal Java SE application, shown in listing 5.7, will be run in a taint-unaware JVM (a) and in an aware environment. On the taint-aware JVM it will be run with (c) and without (b) the selective tainting mechanism enabled. For these three configurations the time needed for computation will be measured. By this, we will be able to see the effect of the selective taint tracking approach proposed in this thesis.

```
1  public class Benchmark {
2      private static String getFoobar() { // this method is declared as source in Juturna's
       configuration, so the returned value is tainted
3          return "foobar";
4      }
5
6      public String foobar = getFoobar().toString();
7      public String str;
8
9      private void run() {
10         for (int i = 0; i < 1e8; i++) {<
11             str = foobar.concat("-").concat(foobar);
12         }
13     }
14
15     public static void main(String[] args) {
16         Benchmark bench = new Benchmark();
17
18         bench.run(); // for warming up the JVM
19
20         long start = System.nanoTime();
21         bench.run(); // this run gets actually measured
22         long end = System.nanoTime();
23
24         System.out.println("Took ms: " + TimeUnit.NANOSECONDS.toMillis(end - start));
25
26         // verificationSink(bench.str); // by adding this line, a flow between source and
           sink will be created and JOANA-Adapter will not consider bench.str as safe
           instance any longer
27     }
28
29     public static void verificationSink(String str) {
30       // dummy, just contained for demonstration purposes
31     }
32 }
```

Listing 5.7.: A minimal Java SE example application used for evaluating the selective taint tracking approach.

For *a* no further preparations had to be made. *b* needs Juturna to be configured appropriately in order to mark the method `getFoobar()` as source. The according (base) configuration is shown in listing 5.9. *c* requires additional preprocessing in the form of analyzing the applications bytecode, in this case a single classfile, performed by the JOANA-Adapter in order to find safe method calls ready to be replaced.

Therefore, the JOANA-Adapter needs to be given a list of sources and sinks via an extra configuration file. This file is shown in listing 5.8. Based on this configuration and the classfile compiled from the code shown in listing 5.7, the JOANA-Adapter computes which method calls' targets might be replaced. These then get added to the configuration file used to run scenario *c* shown in listing 5.10.

Listing 5.8 declares an additional sink to illustrate the concept of selective taint tracking once more. When enabling line 26 in the source of the benchmark appli-

```
# Sources and sinks are listed in this file, following the notation used in they
# the JVM specification: package.StringExample.readPIN()Ljava/lang/String;
#
# Sources
Benchmark.getFoobar()Ljava/lang/String;

# Sinks
Benchmark.verificationSink(Ljava/lang/String;)V
```

Listing 5.8.: Sink and source given to the JOANA-Adapter.

```
{
  "sources": [
    {
      "fqn": "Benchmark.getFoobar()",
      "taintSource": "TAINTED_BY_DEFINITION"
    }
  ],
  "sinks": [
    {
      "fqn": "Benchmark.verificationSink
      (java.lang.String)",
      "parameters": [
        {
          "paramIndex": 0,
          "forbiddenSources": [
            "*"
          ],
          "mitigation": "LOG"
        }
      ]
    }
  ],
  "taintSources": [
    {
      "name": "TAINTED_BY_DEFINITION",
      "severity": "ACTUAL_SOURCE"
    }
  ]
}
```

Listing 5.9.: The base configuration declaring the taint source.

```
{
  "extends": "./juturna.non_selective.config.json",
  "disableTaintingForCalls": {
    "Benchmark.run()": [
      18,
      25
    ]
  }
}
```

Listing 5.10.: Extending the base configuration (on the left) by adding calls to be adjusted.

cation, the calls considered safe by the JOANA-Adapter before are not safe any longer.

Based on the configuration shown in listing 5.10, Juturna's bytecode instrumentation will modify the bytecode of the `run()` method by replacing the two calls pointing to `String#concat()` with ones to `String#___concat()` during the process of classloading as illustrated in figure 5.5.

The results shown in the next subsection have been computed by running every configuration of the benchmark in ten separate invocations of the JVM, always performing one warmup and and one measurement iteration.

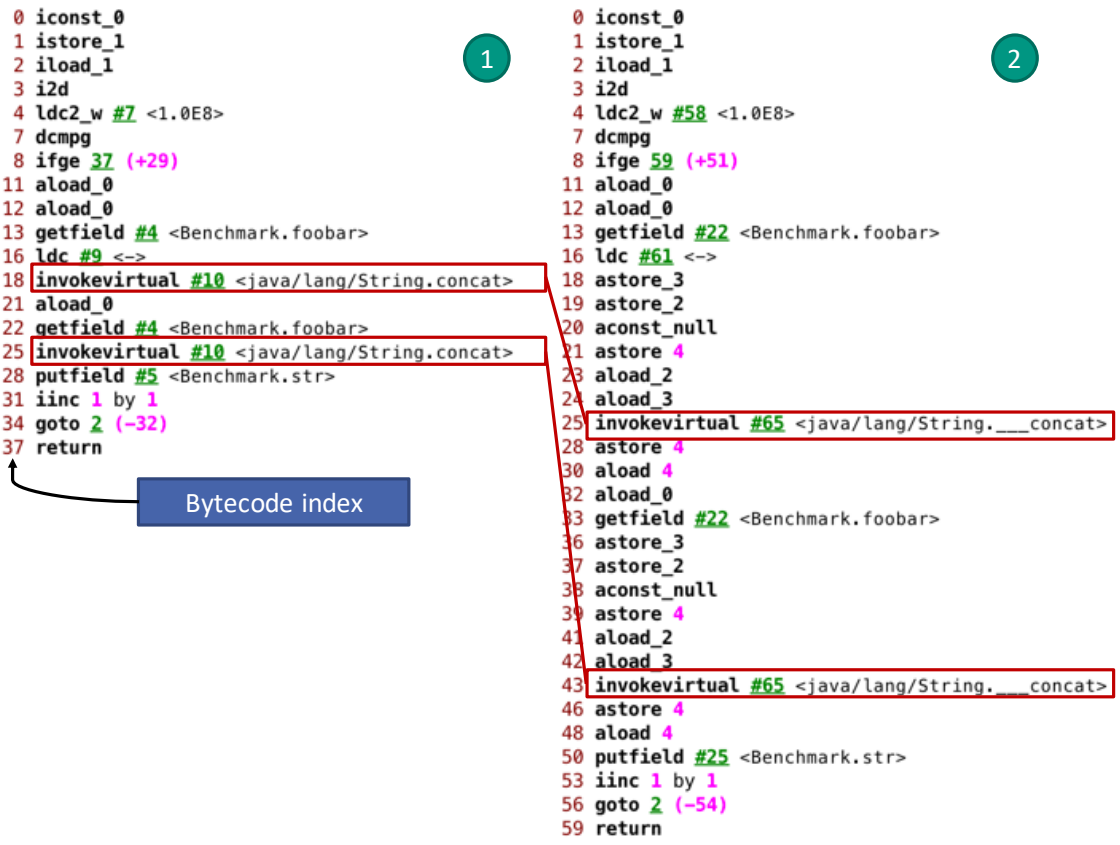

Figure 5.5.: The `SelectiveTaintingInstrumenter` modifies the call instructions considered safe by the JOANA-Adapter in the bytecode as the classfile gets loaded into them JVM.

5.3.2. Results

Figure 5.6 shows the results obtained from this benchmark. The values describe the average time in milliseconds it took to execute the `run()` method. The error bars represent the standard deviation. As reference, the reddish, dotted line represents an approximation of the time the aforementioned naive approach would need assuming a sufficient string length. See the last performance benchmark for a more detailed explanation of this approximation.

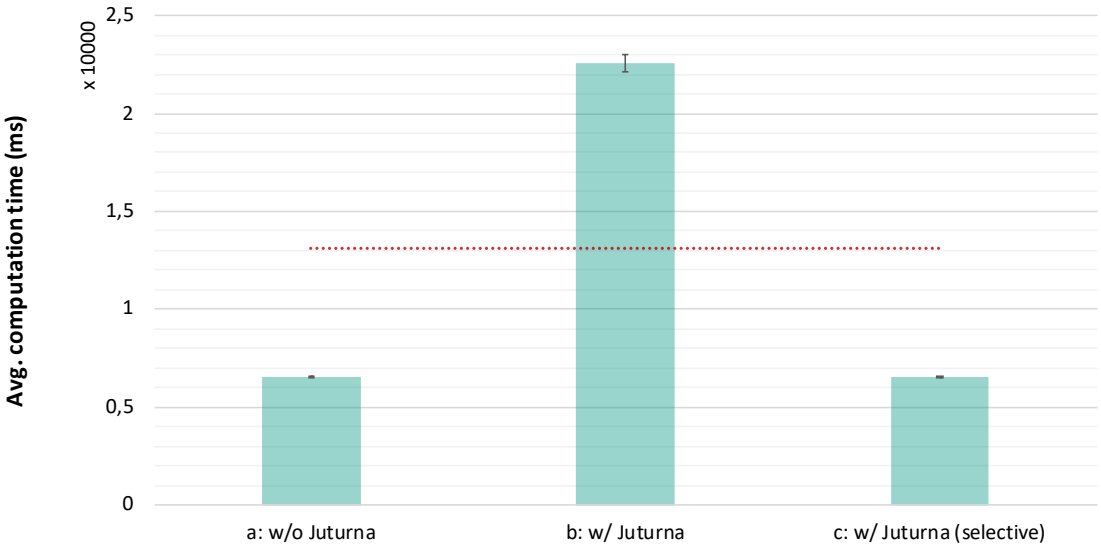

Figure 5.6.: Visualization of the average computation time measured for the three described configurations: a, b and c. A table containing the actual values can be found in the appendix, see table A.3.

Looking at the diagram, one clearly notices the overhead caused by Juturna (b). This is caused, as exhaustively treated in the last section, by focussing on an relatively cheap string manipulation operating on a small string in terms of characters contained.

But of much greater importance is that configuration c took approximately the same time to complete as a did, proving that the idea of selective tainting is – in the best case – capable of completely removing the added overhead. With realistic applications this will not be the case as only a portion of the overhead may be removed because there certainly will be some security-sensitive data flows. But as it will not increase it, there is no risk in applying it[10].

We have seen that for strings whose length grows faster than the amount of attached taint ranges, the overhead becomes less noticeable and so does the effect of selective tainting – but for short ones in applications, or their components, massively manipulating strings having a (clear) segregation between security-sensitive and security-insensitive areas it might be massive.

[10] At least when the analyzed application does not use Java Reflection or similar as they might create dependences not represented in JOANA's SDG leading to an incorrect determination of safe calls.

5.4. Open problems and possible solutions

Beside limitations foreseeable at the very beginning of the work and therefore considered when drafting the concept, some limitations came to light just during the implementation and evaluation phases. As these shall not be swept under the carpet, they will be briefly mentioned (again) subsequently and, as far as possible, strategies to make them less grave will be sketched. Because these suggestions might go beyond the scope of this thesis, they shall be understood as possible future improvements to the ideas and concepts presented in here – which also distinguishes this section from the "outlook" chapter presenting additional features that might be added, not improvements to already existing ones.

The current limitations and weak spots can be split into two major groups: ones related to the (conceptual) capabilities of the taint tracking system itself and ones belonging to the additions making it "selective".

5.4.1. Limitations of string-level taint tracking (and how static checks can help)

As extensively discussed in the concept chapter, taint tracking on a string-level has certain assets and one major drawback: as soon as an implementation decides to operate on the char-level, tracking taint information is not possible any longer. The system is blind to these flows. This limitation is inherent to the concept, but there are ways to mitigate, or at least estimate, the application-specific severity of this "blindness" by using the capabilities of the already included static analysis.

To ensure, such loss of taint information does not occur in the standard library classes themselves, all contained functionality operating on the character-level needs to be augmented. Beside the foundations adjusted so far, there are some more (easy to find) packages/classes requiring to be checked. These include `java.net.UrlEncode`, `java.text.*`, `java.util.Formatter`.

Functionality could be included into the JOANA-Adapter checking whether an application invokes problematic functions like `String#charAt()` or `AbstractStringBuilder#setCharAt()` in code not being provided by the runtime environment and therefore not being augmented already. With precise and clear logs a security engineer might then be able to check these calls, assess them (manually) and act accordingly: e.g., by augmenting them manually or by refraining from operating on such a low level in case performance is not crucial at this spot.

Usage of `char` and `char[]` in application specific code could also be detected easily. This way, the amount of code still needing to be scanned by a developer, in order to assure no injections will happen, can be reduced. Admittedly, it might not go down to zero.

This static check spotting code not playing well with the presented taint tracking system might be extended further to be able to detect the (probably rare) usage of

Java specifics like `String#intern()` (see section 3.5.2). This could either be implemented on top of JOANA in the JOANA-Adapter or as custom configuration/plugin for software like the popular *FindBugs*[11] tool. As the author is aware, this would be another new way of using static analyses in order to create better taint tracking systems.

Another proposal targeting these "dangerous" calls are the already implemented implicit sources, but they are less holistic and may cause noise – and therefore are, admittedly, not usable at all the places where characters "without a history" become part of a string.

5.4.2. Better performance through more efficient taint ranges

As we have seen when discussing the results of the performance benchmark, the decrease in terms of speed when operating on strings with a unfavorable ratio between taint ranges and length is significant. This is because several checks have to be performed when adding another taint range to a `TaintInformation` container holding a sorted list already populated. Depending on the operation performed to the string a lot of ranges might need adjusting: e.g., appending string b to a requires all ranges attached to b to be shifted in order to be appended to the `TaintInformation` container bound to a. All this is expensive – especially in comparison to simple and cheap manipulations on short strings basically just copying characters from one `char[]` to another – although, the algorithms have been thoughtfully implemented.

Therefore, in order to address these performance issues additional effort needs to be spent optimizing the concept and implementation of the taint ranges. One idea is to combine taint ranges with the naive array approach, switching to the latter as soon as the amount of taint ranges reaches a certain threshold[12]. Another very promising approach is the not yet implemented idea of lazy evaluating taint ranges.

Right now, as aforementioned, adding another taint range causes checks and adjustments to be done. But usually taint ranges get added, shifted or overridden way more often than they get evaluated, which basically only happens at sinks. And in case a string does not reach a sink – this actually might be the common case – the taint state is not even retrieved once. Therefore, it seems a good idea to skip some of these checks and adjustments – consciously offending the currently enforced invariants regarding sorting and overlapping – at the time of adding and instead lazily compute the taint state at evaluation time. At retrieval, these "temporary" and "misbehaving" taint ranges can get adjusted in order to fulfill the invariants mentioned, providing a valid taint state. This should help reducing the performance overhead massively. The problem of too many temporary taint ranges attached might be handled by enforcing the taint ranges to be aligned after every k additions.

[11] `http://findbugs.sourceforge.net/`

[12] This would make the system "hybrid" in another sense.

5.4.3. Limitations regarding Java EE

The limitations regarding selective tainting are mostly contained in the JOANA-Adapter – simply because the second step, adjusting the bytecode to call an unaugmented method, is a much simpler task than deciding which calls can be guaranteed to not be invoked on security-sensitive instances of taint-aware types.

Especially when moving from Java SE applications to Java EE web applications, problems arise. JOANA, respectively the underlying WALA project, operates on classfiles, which – at least in the domain of Java SE applications – can be expected to be available before runtime. But with Java EE web applications a lot of logic might be implemented in *JavaServer Pages* (JSP). These are basically HTML pages with Java code embedded in special tags getting compiled to servlets (in the form of JVM bytecode) by the Java EE servlet container directly before being loaded into the JVM. Although it is possible to compile them using tools contained in the *Java Development Kit* (JDK), this makes the process much more difficult as there are further configurations that need to be considered during this step, like libraries introducing additional tags that might be used in the JSP files.

Additionally, Java EE servlet containers tend to introduce additional complexity to the process of classloading in order to provide functionality like automatically reloading servlets as soon as modifications have been detected ("hot swap").

The principle of dependency injection adds further problems: the runtime environment/framework provides the application with an implementation of its choice fulfilling a declared interface, making the application more portable. This is a common pattern used in Java EE applications. In order to make sure JOANA/WALA loads a web application in the same way it would be loaded by the servlet container, resolving references to further classfiles has to happen in the same way – this basically means, that the behavior of the servlet container's classloaders needs to be imitated. The same holds true for the dependency management mechanism. This is possible, but will take some serious amount of effort.

These are mainly problems for static approaches, as taint tracking operates on a lower level not needing to be aware of this – still, Javassist, the library used for adding sources and sinks during classloading by instrumenting the bytecode, also suffers from this. When modifying the bytecode of a classfile it tries to resolve imports in order to know about types declared in other files but used in the current one. This is basically the same problem as described above, but as it occurs at runtime during classloading there should be ways to (simply) reuse the classloaders of the surrounding servlet container. As an alternative, one could replace Javassist with a bytecode manipulation library not needing to look up other classes (e.g., by operating on a lower level without semantic checks).

Another problem is that JOANA does not support Java Reflection and therefore its SDG does not represent any (data) dependencies, new objects, additional implementations of a type, etc., introduced by this technology. Reflection is a massive problem for any static analysis library as it results in runtime behavior not explicitly

represented by opcodes in the bytecode – but there are tools stating to be capable of this, like the static taint analysis tool presented by Huang et al. [48]. In the case of the JOANA-Adapter, this can lead to call nodes being considered safe erroneously – making the whole approach unsound and incorrect. As an example, think of a `StringBuilder` instance to which a tainted string gets appended by calling `append()` via Reflection – the instance is tainted but JOANA would not be able to see this.

As mentioned before, the usage of Reflection is popular in modern frameworks for web applications making this a problem that needs to be resolved by either adding support for handling Reflection to the underlying static analysis toolchain or by having a security engineer manually check that Reflection does not influence the calls determined to be safe.

The bigger an application gets, the bigger the SDG created by JOANA becomes. The complexity of the computations performed by JOANA and the growth of the SDG in combination with limited memory prevents its usage on applications of arbitrary size. The project page states that full Java is supported up to 100kLOC [65].

One last problem to mention is, that JOANA needs to be given an entry point of the application to analyze. As JOANA computes which objects get created and in which order this might happen, no instances should have been created before entering this entry point. With a standard Java SE application this entry point is, per convention, the `main()` method of a stated class. But with Java EE web applications running inside a servlet container, this entry point may be in the container itself as it receives the actual requests and then delegates them to the various servlets of the web application. To be brief – with Java EE and some frameworks one can create almost arbitrarily complex set-ups.

These problems and the coupled difficulties are not the fault of JOANA, it is simply the complexity caused by patterns common in the domain of Java EE making it not the ideal area of application for static analysis mechanisms. They have not been designed for such scenarios and work great for smaller, standalone components. Dependency injection, Reflection, etc. are not transparent for static analysis and need plenty of adjustments[13] – but they are transparent for the taint tracking mechanism of Juturna, causing only minor problems.

But also the taint tracking prototype implemented so far suffers from the enormous complexity introduced by modern web frameworks like Spring. Nowadays, actual code written gets abstracted from the Java Servlet Specification, e.g., by using annotations in order to declare HTTP endpoints, making it much more difficult to annotate sources of taint. Simply sticking to the ones derived from the specifi-

[13] Huang et al. describe Reflection and (web) frameworks as “the bane of static taint analysis” [48].

cation, which are hidden inside the framework, does not work well in all cases as the framework might apply some character-level operations to the values retrieved from them: automatically parsing and converting a received JSON payload to a Java object, parsing URL parameters, etc..

When evaluating whether weaknesses in web applications can be found using Juturna in section 5.1, the problem of precisely addressing a sink arose. In that case, the Java Servlet Specification stated that HTTP responses need to send by calling `write()` on an instance of `PrintWriter`, which is a popular class extended by many classes used in completely different contexts, received by calling `ServletResponse.getWriter()`. Using abstract types and interfaces for the specification of APIs is considered a good practice in software development – but in this context it causes problems. Declaring a method on a popular supertype as "inheriting sink" without unintentionally instrumenting the method on subtypes, but without being to specific and possibly missing other implementations due to polymorphism requires some knowledge about the used framework/servlet container. Both problems apply to sources as well as sinks.

A possible solution is, as already mentioned when introducing source level augmentation, to analyze the behavior of a framework/servlet container and to augment its code, adding source code marking strings as tainted at the appropriate places. This basically follows the principle of augmentation applied to the standard library.

Another suggestion therefore is to rethink the concept of (declaring) sources and sinks and making it more flexible. A very powerful enhancement would be to support sources not returning an instance of a taint-aware type, e.g., an object encapsulating a string or, referring to the problem above, a `PrintWriter` instance, in a way to wrap the returned object by a proxy created via Reflection. This proxy would than pass through all calls and field accesses – therefore able to take care of tainting returned values and taint checking values given as parameters without knowing which concrete (sub)type actually has been wrapped.

This problem of precisely marking sinks and sources in complex systems has, to the knowledge of the author, not been handled by any of the publications describing similar systems so far.

6. Conclusion

6.1. Summary

In this thesis, a pluggable, lightweight and selective taint tracking system for Java has been drafted, implemented and evaluated. Motivated by the imminent danger coming from injection attacks and the harm they might cause – to companies and individuals – and former research pointing out the applicability of taint tracking regarding the detection of attacks belonging to this class, the taint tracking system Juturna has been designed and prototyped. It tries to provide *Runtime Application Self-Protection* for real Java (EE) (web) applications in the long term with injection attacks and business requirements in mind.

Juturna employs the well-known dynamic analysis of taint tracking, following concepts presented by other researchers while often enhancing them, beside also introducing many new ideas. Taint tracking immanently causes a runtime overhead – regarding computational and memory requirements. Juturna tries to reduce them by applying the concept of "taint ranges" – novel to the domain of Java taint tracking – instead of the common "shadow arrays" for a reduced memory usage when attaching taint information to strings. It also adds more flexibility regarding the kind and amount of metadata that can possibly be attached. Additionally, a new take on the integration of static, PDG-based IFC analysis mechanisms makes Juturna a hybrid system using static and dynamic IFC techniques – but with a strong focus on dynamic taint tracking.

Analyzing the usage scenario and the way injection attacks work, tracking only string-like data seemed to best fit the requirements: potentially capable of capturing the vast majority of injection attacks while keeping the to be expected overhead within reasonable bounds. The issues resulting from this choice have been discussed and countermeasures have been suggested. We saw the unfavorable imprecisions/false-positives tainting with a string-granularity might result in, therefore deciding to implement an approach with character-granularity.

From several possible implementation strategies for the core of the taint tracking system, augmenting the source code of the Java standard library has been chosen. The string-like classes, `java.lang.String`, `java.lang.StringBuilder` and `java.lang.StringBuffer` have been given a new field representing their respective taint state. Additionally, the methods of these classes, among others, have been

enhanced/augmented in order to correctly propagate the taint information along during the application's execution. This approach enables the usage of taint ranges, is easy to maintain and very portable due to the fact that the JVM used for running applications does not need to be modified. This is done by "injecting" a modified version of the standard library.

Beside the advantages of this technique, we also discussed its downsides – most noteworthy the missing ability to track taint for primitive `char`s or arrays containing them and risk of losing taint information this way.

Additionally, a bytecode instrumentation component is included in Juturna in order to adjust declared sources and sinks in an application's bytecode on-the-fly without the need to modify its source code. Juturna therefore fulfills the formulated requirements regarding portability and being pluggable. The secondary goals of being configurable, extendable and easy to maintain have been present during the whole process of sketching and implementing and have been met.

Motivating the idea of selective taint tracking, it has been worked out that many information flows in an application are actually not security-sensible by either not being influenced by a source or by not flowing into a sink – and therefore do not need to be tracked. By including a preprocessing step utilizing static analysis mechanisms, these are determined and tracking instructions can be removed by another on-the-fly bytecode manipulation. Although there has been research done before regarding hybrid approaches, Juturna shows a new way of including static analysis helping the dynamic taint tracking analysis to be more efficient.

In three evaluation scenarios the prototype of Juturna has been evaluated regarding its ability to find weaknesses promoting reflected XSS attacks in small Java EE servlets, the performance of the core taint tracking system and the basic operability of the selective taint tracking extension. The prototype was able to detect the flaws and to interoperate with an Apache Tomcat as Java EE servlet container and it showed a reasonable performance compared to an approximated baseline implementation using the naive concept of arrays for storing taint information. In addition, we saw the potential of the hybrid tainting approach – even though the prototype's capabilities are yet limited.

Although many issues and ideas for improvements came up during this phase, the evaluation's results are quite promising altogether. Even some first, positive feedback has been received from a team at SAP working on the *Cloud Platform* solution[1], interested in integrating the results of this work into their product with the goal of developing it further in order to provide enhanced security features to their customers. This shows, the project is headed in the right direction.

[1]SAP Cloud Platform is a PaaS providing, among others, an environment compliant with the Java EE web profile.

6.2. Discussion

During drafting and implementation, many decisions had to be taken determining the further course of the project. After finishing the work on a first prototype, and with some evaluations performed, it is time to look at those in hindsight.

The main characteristics of a taint tracking system are *how* it tracks *which* information on *what* granularity level. Deciding between a bytecode instrumentation approach or the, ultimately chosen, augmentation on source level was not easy as both have their respective advantages and disadvantages.

In retrospect, using bytecode instrumentation not only for explicit sources, sinks and sanitization functions, but for the whole taint tracking system would eliminate the need to manually adjust all methods of the standard library[2] operating on primitive `char`s. But as this is basically a onetime task[3] it should be achievable by a team of developers in short time, depending on the complexity and amount of libraries/frameworks linked to an application. The ground work has already been done.

On the other hand, a bytecode instrumentation would be a much more generic, "steamroller tactic" not tacking available context information into account. Additionally, the concept of taint ranges could hardly be applied then.

In the given scenario, tracking taint information only for strings, one therefore can consider augmentation to be the better technique – also in hindsight. Especially, once it will be equipped with a static code check mechanism raising a developer's attention in case of problematic code, e.g. `charAt()`, spotted (see section 5.4.1). Although it is justified to assume, only usage in real-world scenarios can show whether string-level granularity is sufficient and whether the trade-off between the limitation on tracked data types and the reduced overhead caused will pay off.

As extensively discussed in section 5.4.3, static IFC – and therefore also the selective approach presented – massively struggles with the complexity and flexibility introduced by modern web frameworks. Taint tracking is less affected by this, as most of the mechanism problematic for its static companion are transparent to it (e.g., Java Reflection, dependency injection). But beside all the problems introduced by modern web frameworks regarding the detection of injection attacks, one should not forget that they already fix a lot of vulnerabilities by adding validation, escaping and enforcing the observance of some common patterns for more readable and maintainable code.

In the preceding evaluation, benchmarking Juturna's taint tracking mechanism, a significant overhead could clearly be measured – but in a kind of worst-case sce-

[2] In a bussiness scenario, some project dependencies, e.g., libraries and frameworks, probably also need to be taken care of.

[3] Updates to the standard library classes of Java are expected to, if at all, only add functionality. Updates to libraries might require some more adjustments.

nario for the system because all the benchmark did was manipulating strings. As explained, the overhead can be expected to be much less striking in real-world applications performing not only string manipulations.

This leads to the question whether the effort needed to enhance the successfully tested selective extension in order to get over all the (unexpected) problems arose in the context of Java EE, is worth the decreased, but presumably already very low, overhead in real-world settings. For string-only tracking the answer to this question might possibly be "no". Taking the idea itself and combining it with another taint tracking system, not only looking at strings and therefore causing a much bigger overhead, the answer might be "yes". But in order to properly answer this, the assumptions regarding Juturna's real-world overhead needs to be verified first.

But, and this can be seen as an advantage over the hybrid approaches of Mongiovì et al. [45] and Zhao et al. [46], the selective extension does not have to be applicable to a (complex) application in order for the dynamic taint tracking to work, as they have been, on purpose, very loosely coupled. In case an application massively uses techniques like Reflection, the preceding analysis might simply be skipped. Additionally, it should be possible to only use the selective mechanism for isolated, performance-critical components in such a web project.

Nevertheless, for "general-purpose" taint tracking systems not focussing on strings only, one can firmly assume that hybrid approaches, combining the best of both worlds, are the way to go in order to get the best results both taking detection rates and performance into account.

6.3. Outlook

A lot of suggestions on how to improve the current prototype of Juturna have been made already when discussing open issues and possible solutions in section 5.4. Especially assistance on spotting code operating on primitive `char`s and the described improvements towards the concept of taint ranges seem very promising and should be tackled in followup projects.

Additionally, in order to improve the applicability of Juturna in the context of complex web applications, the mechanism for declaring sources and sinks should be enhanced and a comprehensive base configuration declaring the sources and sinks relevant for Java EE servlets should be provided.

Beside these improvements, enhancing already existing functionality, further extensions could be made.

Such extensions could possibly be to make taint information persistent by, e.g., not only storing values but also taint information in a database. A simple approach towards this would be to adjust the database driver in Java in order to have it rewrite queries before submitting them. By this, taint information could then be stored in "shadow columns" or "shadow tables", at least partially transparent to not

taint-aware other applications.

This would be a step towards inter-system, inter-platform taint tracking as different systems would be able to exchange not only data, but also the attached taint information. Taking this idea further, also HTTP could be leveraged to not only transport data between a browser and a web server, or between micro services, but also taint information.

Such an "end-to-end" approach could make detection of reflected XSS more precise – as sinks inside a browser could be used for checking instead of marking the HTTP response as sink – avoiding "false-positives"[4]. The browser could then decide whether this is harmful or not. Being able to persist taint information, this can also be extended to the detection of persistent XSS.

It would combine different applications and platforms into one big, abstract system covered by a virtual, highly dynamic taint tracking system consisting of multiple ones interoperating under the hood. This would not be feasible using purely static IFC based approaches – but they might be integrated as part of a hybrid approach.

First preliminary investigations regarding persistent and end-to-end taint tracking in the context of adjusted, taint-aware versions of *Node.js*, Mozilla Firefox and SQL-databases have been done by the cooperating department at SAP under the lead of Dr. Martin Johns.

It could also be evaluated for which other fields of application the taint tracking system developed might be used, respectively whether its capabilities would be sufficient for use cases like the assertion of access permissions or the enforcement of privacy policies restricting the usage of an individual's personal information, e.g., within a system for *Customer Relationship Management (CRM)*, in the context of data privacy.

Irrespective of followup extensions and improvements to Juturna, my conclusion regarding the usage of IFC mechanisms for the detection of injection vulnerabilities and my estimate regarding its future development are as follows: IFC seems to be the perfect solution for this type of problem. There is no other approach for the detection of this broad class of partly complex weaknesses being as qualified as IFC – even so both dynamic and static approaches have drawbacks, especially when dealing with modern web applications. But these can be, at least partially, compensated by combined, hybrid approaches. There is still a lot of work to do, but I am sure that hybrid approaches building up on the insights presented in this work are the future.

[4]Following the definition of a malicous flow given earlier, those would not be false-positives as there is an actual flow from a source to a sink. But, with further context-specific knowledge and more specific sinks and sources, more precise real-world detection results can be achieved.

A. Appendix

The image on the title page is public domain and has been taken from `https://www.pexels.com/photo/aroma-aromatic-bean-beans-261463/`.

Benchmark (task)	Configuration	Avg. ops/s	Std. deviation	SF
a	w/o Juturna: k=0	5 901 283.12	189 891.34	1.000
	k=0	6 095 377.79	159 453.71	0.968
	k=1	2 904 112.00	47 739.55	2.032
	k=3	2 549 839.14	37 814.54	2.314
	k=10	1 793 320.11	23 219.89	3.291
	k=100	304 848.51	6 281.16	19.358
b	w/o Juturna: k=0	35 246 220.96	491 093.15	1.000
	k=0	33 608 075.49	1 605 776.05	1.049
	k=1	3 382 186.49	163 048.06	10.421
	k=3	3 010 376.46	38 737.98	11.708
	k=10	2 004 764.66	55 672.52	17.581
	k=100	317 842.93	5 258.20	110.892
c	w/o Juturna: k=0	57 477 632.43	1 918 390.73	1.000
	k=0	54 631 253.91	893 137.95	1.052
	k=1	6 321 667.11	85 144.57	9.092
	k=3	5 663 686.09	136 420.19	10.148
	k=10	4 627 192.62	74 310.66	12.422
	k=100	944 783.87	10 789.87	60.837
d	w/o Juturna: k=0	3 562 222.76	70 092.24	1.000
	k=0	3 776 332.18	61 439.52	0.943
	k=1	2 743 547.55	42 430.86	1.298
	k=3	2 556 040.12	33 835.66	1.394
	k=10	2 054 137.24	27 331.60	1.734
	k=100	365 662.08	12 236.93	9.742

Table A.1.: Average throughput in operations per second with $n = 250$. SF describes the slowdown factor of a given configuration compared to the baseline (w/o Juturna: k = 0).

Benchmark (task)	Configuration	Avg. ops/s	Std. deviation	SF
a	w/o Juturna: k=0	186 854.63	2 020.05	1.000
	k=0	187 033.04	1 994.16	0.999
	k=1	178 404.95	1 420.43	1.047
	k=3	177 778.85	2 874.39	1.051
	k=10	169 396.85	2 154.92	1.103
	k=100	118 026.72	1 109.23	1.583
b	w/o Juturna: k=0	2 394 708.04	26 689.62	1.000
	k=0	2 455 809.99	34 757.04	0.975
	k=1	1 497 452.75	71 775.52	1.599
	k=3	1 398 813.64	29 801.17	1.712
	k=10	1 150 497.05	40 313.69	2.081
	k=100	293 841.86	3 920.34	8.150
c	w/o Juturna: k=0	4 659 760.11	67 221.49	1.000
	k=0	4 894 947.01	37 337.28	0.952
	k=1	2 807 368.73	28 983.78	1.660
	k=3	2 678 625.75	36 349.81	1.740
	k=10	2 426 939.25	33 335.55	1.920
	k=100	809 156.24	9 706.05	5.759
d	w/o Juturna: k=0	151 423.83	1 911.47	1.000
	k=0	150 578.89	1 598.92	1.006
	k=1	144 188.66	4 886.38	1.050
	k=3	147 471.60	6 324.63	1.027
	k=10	142 843.56	6 091.06	1.060
	k=100	111 390.12	1 374.05	1.359

Table A.2.: Average throughput in operations per second with $n = 10^4$. SF describes the slowdown factor of a given configuration compared to the baseline (w/o Juturna: k = 0).

	Avg. computation time (ms)	Std. deviation	SF
W/o Juturna	6 552.900	58.966	1.000
W/ Juturna	22 587.960	427.898	3.447
W/ Juturna (selective)	6 519.500	50.871	0.995

Table A.3.: Computation times in milliseconds measured and averaged over 10 runs for three different configurations: Java without Juturna, Java with Juturna and Java with selective taint tracking. SF describes the slowdown factor of a given configuration compared to the baseline (w/o Juturna).

```java
@Warmup(iterations = 5)
@Fork(5)
@BenchmarkMode(Mode.Throughput)
@Measurement(iterations = 5)
@State(Scope.Benchmark)
public class JuturnaBenchmark {
  @Param({"250", "10000"})
  static int LEN;

  @Param({"0", "1", "3", "10", "100"})
  static int TAINT_RANGES;

  String str_a;
  StringBuilder sb_b;
  StringBuilder sb_c;
  StringBuilder sb_d;

  @Setup
  public void setup() {
    str_a = getInitialString(TAINT_RANGES).toString();
    sb_b = getInitialString(TAINT_RANGES);
    sb_c = getInitialString(TAINT_RANGES);
    sb_d = getInitialString(TAINT_RANGES);
  }

  private static StringBuilder getInitialString(int containedTaintRanges) {
    StringBuilder sb = new StringBuilder();

    for (int i = 0; i < LEN; i++) {
      sb.append("a");
    }

    if (THelper.isTaintAwareContext() && containedTaintRanges > 0) {
      TaintInformation ti = THelper.get(sb);
      for (int i = 0; i < containedTaintRanges; i++) {
        int start = sb.length() - containedTaintRanges + i;
        ti.addRange(start, start + 1, (short) i);
      }
    }

    return sb;
  }

  @Benchmark
  public void a() {
    str_a = str_a.substring(0, LEN / 2).concat(str_a.substring(LEN / 2));
  }

  @Benchmark
  public void b() {
    sb_b.insert(LEN / 2, "foobar");
    sb_b.delete(LEN / 2, LEN / 2 + 6);
  }

  @Benchmark
  public void c() {
    sb_c.replace(LEN / 2, LEN / 2 + 6, "foobar");
  }

  @Benchmark
  public void d() {
    sb_d.reverse().reverse();
  }
}
```

Listing A.1.: The complete source of the benchmark discussed in section 5.2.2. Still, supplementary code, like assertions in the "tear-down" phase ensuring taint information has been propagated correctly during the benchmark, has not been printed.

List of figures

List of listings

List of tables

List of abbreviations

AOP	*Aspect Oriented Programming* . . . 18
CWE	*Common Weakness Enumeration* . . . 6
CWE	*Common Weakness Enumeration* . . . 6
FQN	*Fully Qualified Name* . . . 4
GPL	*GNU General Public License* . . . 53
IFC	*Information Flow Control* . . . 41
JDK	*Java Development Kit* . . . 102
JMH	*Java Microbenchmarking Harness* . . . 87
JRE	*Java Runtime Environment* . . . 3
JSON	*JavaScript Object Notation* . . . 62
JSP	*JavaServer Pages* . . . 102
JVM	*Java Virtual Machine* . . . 5
OWASP	*Open Web Application Security Project* . . . 1
PaaS	*Platform as a Service* . . . 24
PDG	*Program Dependence Graph* . . . 12
RASP	*Runtime Application Self-Protection* . . . 16
SDG	*System Dependence Graph* . . . 43
VCS	*Version Control System* . . . 45
XSS	*Cross-site Scripting* . . . 1

Literature & References

[1] The Open Web Application Security Project (OWASP). *Unvalidated Input*. 22 April 2010. URL: https://www.owasp.org/index.php?title=Unvalidated_Input&oldid=82263 (visited on 12/12/2017).

[2] The Open Web Application Security Project (OWASP). *OWASP Top 10 - 2017 RC2*. October 2017. URL: https://github.com/OWASP/Top10/raw/master/2017/OWASP%20Top%2010%202017%20RC2%20Final.pdf.

[3] Sebastian Lekies, Ben Stock and Martin Johns. "25 million flows later: large-scale detection of DOM-based XSS". In: *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. ACM. 2013, pp. 1193–1204.

[4] Vivek Haldar, Deepak Chandra and Michael Franz. "Dynamic taint propagation for Java". In: *Proceedings - Annual Computer Security Applications Conference, ACSAC*. 2005, pp. 303–311. DOI: 10.1109/CSAC.2005.21.

[5] William G. J. Halfond and Alessandro Orso. "AMNESIA: Analysis and Monitoring for NEutralizing SQL-injection Attacks". In: *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*. ASE '05. Long Beach, CA, USA: ACM, 2005, pp. 174–183. DOI: 10.1145/1101908.1101935.

[6] Wikipedia. *Juturna — Wikipedia, The Free Encyclopedia*. URL: https://en.wikipedia.org/w/index.php?title=Juturna&oldid=778284943 (visited on 18/11/2017).

[7] Oracle. *Differences between Java EE and Java SE - Your First Cup: An Introduction to the Java EE Platform*. 2012. URL: https://docs.oracle.com/javaee/6/firstcup/doc/gkhoy.html (visited on 23/01/2018).

[8] Oracle. *Java Platform, Enterprise Edition: The Java EE Tutorial, Overview*. 2014. URL: https://docs.oracle.com/javaee/7/tutorial/overview.htm (visited on 23/01/2018).

[9] David Delabassee. *Opening Up Java EE - An Update*. 12 September 2017. URL: https://blogs.oracle.com/theaquarium/opening-up-ee-update (visited on 23/01/2018).

[10] Shing Wai Chan and Ed Burns. *Java Servlet Specification, Version 4.0*. July 2017. URL: https://javaee.github.io/servlet-spec/downloads/servlet-4.0/servlet-4_0_FINAL.pdf.

[11] Oracle. *Your First Cup: An Introduction to the Java EE Platform*. 2017. URL: https://javaee.github.io/firstcup/java-ee002.html (visited on 24/01/2018).

[12] The Common-Weakness-Enumeration Project. *CWE-74: Improper Neutralization of Special Elements in Output Used by a Downstream Component ('Injection')*. 5 May 2017. URL: https://cwe.mitre.org/data/definitions/74.html.

[13] The Common-Weakness-Enumeration Project. *CWE-20: Improper Input Validation*. 5 May 2017. URL: https://cwe.mitre.org/data/definitions/20.html.

[14] William G J Halfond, Alessandro Orso and Panagiotis 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*. 2006, p. 175. DOI: 10.1145/1181775.1181797.

[15] The Common-Weakness-Enumeration Project. *CWE-79: Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')*. 5 May 2017. URL: http://cwe.mitre.org/data/definitions/79.html.

[16] The CERT Project at the Carnegie Mellon University. *Malicious HTML Tags Embedded in Client Web Requests*. 3 February 2000. URL: https://web.archive.org/web/20171012075417/https://www.cert.org/historical/advisories/CA-2000-02.cfm.

[17] The MDN Web Docs Project. *Same-origin policy*. 28 August 2017. URL: https://web.archive.org/web/20171113104038/https://developer.mozilla.org/de/docs/Web/Security/Same-origin_policy.

[18] The Web Application Security Consortium. *Cross Site Scripting*. April 2017. URL: http://web.archive.org/web/20170831212314/http://projects.webappsec.org/w/page/13246920/Cross%20Site%20Scripting.

[19] Amit Klein. *DOM Based Cross Site Scripting or XSS of the Third Kind*. Web Application Security Consortium. 4 July 2005. URL: http://web.archive.org/web/20171109095307/http://www.webappsec.org/projects/articles/071105.shtml.

[20] Eric Lai. *Teen uses worm to boost ratings on MySpace.com*. Computerworld.com. 17 October 2005. URL: https://web.archive.org/web/20160402234525/http://www.computerworld.com/article/2558730/malware-vulnerabilities/teen-uses-worm-to-boost-ratings-on-myspace-com.html.

[21] Daniel Hedin and Andrei Sabelfeld. "A Perspective on Information-Flow Control". In: *Proceedings of the 2011 Marktoberdorf Summer School* (2011). URL: http://www.cse.chalmers.se/%7B~%7Dandrei/mod11.pdf.

[22] Christian Hammer and Gregor Snelting. “Flow-sensitive, Context-sensitive, and Object-sensitive Information Flow Control Based on Program Dependence Graphs”. In: *Int. J. Inf. Secur.* 8.6 (October 2009), pp. 399–422. DOI: 10.1007/s10207-009-0086-1.

[23] Dorothy E. Denning and Peter J. Denning. “Certification of programs for secure information flow”. In: *Communications of the ACM* 20 (1977), pp. 504–513. DOI: 10.1145/359636.359712.

[24] Jeanne Ferrante, Karl J. Ottenstein and Joe D. Warren. “The Program Dependence Graph and Its Use in Optimization”. In: *ACM Trans. Program. Lang. Syst.* 9.3 (July 1987), pp. 319–349. DOI: 10.1145/24039.24041.

[25] Jürgen Graf et al. “Tool Demonstration: JOANA”. In: *Principles of Security and Trust - 5th International Conference, POST 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings*. Ed. by Frank Piessens and Luca Viganò. Vol. 9635. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2016, pp. 89–93. DOI: 10.1007/978-3-662-49635-0_5.

[26] Jürgen Graf. “Speeding Up Context-, Object- and Field-Sensitive SDG Generation”. In: *Proceedings of the 2010 10th IEEE Working Conference on Source Code Analysis and Manipulation*. SCAM ’10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 105–114. ISBN: 978-0-7695-4178-5. DOI: 10.1109/SCAM.2010.9.

[27] Y. N. Srikant and P. Shankar. *The compiler design handbook optimizations and machine code generation*. CRC Press, 2008. ISBN: 9781420043839.

[28] Susan Horwitz, Thomas Reps and David Binkley. “Interprocedural Slicing Using Dependence Graphs”. In: *ACM Trans. Program. Lang. Syst.* 12.1 (January 1990), pp. 26–60. DOI: 10.1145/77606.77608.

[29] Kangkook Jee, Georgios Portokalidis and Vp. Kemerlis. “A General Approach for Efficiently Accelerating Software-based Dynamic Data Flow Tracking on Commodity Hardware”. In: *Proceedings of the 19th Internet Society (ISOC) Symposium on Network and Distributed System Security (NDSS)* (2012). URL: http://www.cs.columbia.edu/nsl/papers/2012/tfa.ndss12.pdf.

[30] JC Martinez Santos, Yunsi Fei and ZJ Shi. “Static secure page allocation for light-weight dynamic information flow tracking”. In: *Proceedings of the 2012 international …* (2012), pp. 27–36. DOI: 10.1145/2380403.2380415.

[31] Jonathan Bell and Gail Kaiser. “Dynamic Taint Tracking for Java with Phosphor (Demo)”. In: (2015), pp. 409–413. DOI: 10.1145/2771783.2784768.

[32] Edward J Schwartz et al. “All You Ever Wanted to Know About Dynamic Taint Analysis Forward Symbolic Execution (but might have been afraid to ask)”. In: (2010), pp. 1–5. URL: https://users.ece.cmu.edu/%7B~%7Daavgerin/papers/Oakland10.pdf.

[33] R Sekar. *An Efficient Black-box Technique for Defeating Web Application Attacks*. January 2009. URL: http://www.isoc.org/isoc/conferences/ndss/09/pdf/02.pdf.

[34] Nathan Patwardhan, Ellen Siever and Stephen Spainhour. *Perl in a Nutshell: A Desktop Quick Reference (In a Nutshell (O'Reilly))*. O'Reilly Media, 2002. ISBN: 0-596-00241-6.

[35] James Clause, Wanchun Li and Alessandro Orso. "Dytan: a generic dynamic taint analysis framework". In: *Proceedings of the 2007 international symposium on Software testing and analysis* (2007), pp. 196–206. DOI: 10.1145/1273463.1273490.

[36] William Enck et al. "TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones". In: *ACM Trans. Comput. Syst. ACM Transactions on Computer Systems* 5 (). DOI: 10.1145/2619091.

[37] Lukas Weichselbaum et al. "Andrubis: Android malware under the magnifying glass". In: *Vienna University of Technology, Tech. Rep. TR-ISECLAB-0414-001* (2014).

[38] Deepak Chandra and Michael Franz. "Fine-Grained Information Flow Analysis and Enforcement in a Java Virtual Machine". In: *Twenty-Third Annual Computer Security Applications Conference (ACSAC 2007)*. 2007, pp. 463–475. DOI: 10.1109/ACSAC.2007.37.

[39] Srijith K Nair et al. "A Virtual Machine Based Information Flow Control System for Policy Enforcement". In: *Electronic Notes in Theoretical Computer Science* (2007). URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.76.3914%7B%5C&%7Drep=rep1%7B%5C&%7Dtype=pdf.

[40] Erika Chin and David Wagner. "Efficient character-level taint tracking for Java". In: *Proceedings of the 2009 ACM workshop on Secure web services*. ACM, 2009, pp. 3–12. URL: http://dl.acm.org/citation.cfm?id=1655125.

[41] Christof Dallermassl. *Dynamic Security Taint Propagation in Java via Java Aspects*. URL: https://github.com/cdaller/security_taint_propagation (visited on 16/05/2017).

[42] Omer Tripp et al. "ANDROMEDA: Accurate and Scalable Security Analysis of Web Applications". In: *Proceedings of the 16th International Conference on Fundamental Approaches to Software Engineering*. FASE'13. Rome, Italy: Springer-Verlag, 2013, pp. 210–225. DOI: 10.1007/978-3-642-37057-1_15.

[43] Ashish Aggarwal and Pankaj Jalote. "Integrating Static and Dynamic Analysis for Detecting Vulnerabilities". In: *30th Annual International Computer Software and Applications Conference (COMPSAC'06)* (2006), pp. 343–350. DOI: 10.1109/COMPSAC.2006.55.

[44] Walter Chang, Brandon Streiff and Calvin Lin. “Efficient and extensible security enforcement using dynamic data flow analysis”. In: *Proceedings of the 15th ACM conference on Computer and communications security - CCS ’08* (2008), p. 39. DOI: 10.1145/1455770.1455778.

[45] M. Mongiovì et al. “Combining static and dynamic data flow analysis : a hybrid approach for detecting data leaks in Java applications”. In: *Proceedings of the 30th Annual ACM Symposium on Applied Computing*. 2015, pp. 1573–1579. DOI: 10.1145/2695664.2695887.

[46] Jingling Zhao et al. “Dynamic taint tracking of web application based on static code analysis”. In: *Proceedings - 2016 10th International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing, IMIS 2016* (2016), pp. 96–101. DOI: 10.1109/IMIS.2016.46.

[47] Mark Weiser. “Program Slicing”. In: *Proceedings of the 5th International Conference on Software Engineering*. ICSE ’81. San Diego, California, USA: IEEE Press, 1981, pp. 439–449. URL: http://dl.acm.org/citation.cfm?id=800078.802557.

[48] Wei Huang, Yao Dong and Ana Milanova. “Type-based taint analysis for Java web applications”. In: *International Conference on Fundamental Approaches to Software Engineering*. Springer. 2014, pp. 140–154.

[49] D. Dolev and A. Yao. “On the security of public key protocols”. In: *IEEE Transactions on Information Theory* 29.2 (March 1983), pp. 198–208. DOI: 10.1109/TIT.1983.1056650.

[50] R. Fielding and J. Reschke. *Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content*. RFC 7231. June 2014. URL: http://www.rfc-editor.org/rfc/rfc7231.txt.

[51] Fabien A. P. Petitcolas. “Kerckhoffs’ Principle”. In: *Encyclopedia of Cryptography and Security*. Ed. by Henk C. A. van Tilborg and Sushil Jajodia. Boston, MA: Springer US, 2011, pp. 675–675. DOI: 10.1007/978-1-4419-5906-5_487.

[52] Boze Zekan, Mark Shtern and Vassilios Tzerpos. “Protecting Web Applications via Unicode Extension”. In: (2015). URL: http://ieeexplore.ieee.org/ielx7/7066219/7081802/07081852.pdf?tp=%7B%5C&%7Darnumber=7081852%7B%5C&%7Disnumber=7081802.

[53] David Flanagan. *Java In A Nutshell, 5th Edition*. O’Reilly Media, 2005. ISBN: 0596007736.

[54] James Gosling et al. *The Java® Language Specification - Java SE 8 Edition*. 13 February 2015. URL: https://docs.oracle.com/javase/specs/jls/se8/jls8.pdf (visited on 19/12/2017).

[55] Stephan Pfistner. “End-To-End Taint Tracking In The Web”. TU Darmstadt and SAP SE. 14 October 2015. Unpublished master thesis.

[56] Daniel Jackson and Eugene J. Rollins. “A New Model of Program Dependences for Reverse Engineering”. In: *Proceedings of the 2Nd ACM SIGSOFT Symposium on Foundations of Software Engineering*. SIGSOFT ’94. New Orleans, Louisiana, USA: ACM, 1994, pp. 2–10. DOI: 10.1145/193173.195281.

[57] Thomas Reps and Genevieve Rosay. “Precise Interprocedural Chopping”. In: *Proceedings of the 3rd ACM SIGSOFT Symposium on Foundations of Software Engineering*. SIGSOFT ’95. Washington, D.C., USA: ACM, 1995, pp. 41–52. DOI: 10.1145/222124.222138.

[58] Robert C. Martin. *Clean Code – Deutsche Ausgabe*. mitp-Verlag, 2009. ISBN: 3826655486.

[59] The OpenJDK Project / Oracle. *GNU General Public License, version 2, with the Classpath Exception*. URL: http://openjdk.java.net/legal/gplv2+ce.html (visited on 01/09/2017).

[60] mygnulinux.com. *The only major threat to open source software license models like the GPL is the spread of ‘cloud computing’ and Software as a Service (SaaS) business models*. URL: http://www.mygnulinux.com/?page_id=866 (visited on 01/09/2018).

[61] Unicode. *Unicode Character Database – special casing*. 14 April 2017. URL: https://unicode.org/Public/UNIDATA/SpecialCasing.txt (visited on 12/01/2018).

[62] Christian Ullenboom. *Java ist auch eine Insel Einführung, Ausbildung, Praxis*. Bonn: Rheinwerk, 2016. ISBN: 978-3836241199.

[63] Edsger W. Dijkstra. “The Humble Programmer”. In: *Commun. ACM* 15.10 (October 1972), pp. 859–866. DOI: 10.1145/355604.361591.

[64] Andy Georges, Dries Buytaert and Lieven Eeckhout. “Statistically Rigorous Java Performance Evaluation”. In: *Proceedings of the 22Nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications*. OOPSLA ’07. Montreal, Quebec, Canada: ACM, 2007, pp. 57–76. DOI: 10.1145/1297027.1297033.

[65] Lehrstuhl für Programmierparadigmen - IPD Snelting. *JOANA (Java Object-sensitive ANAlysis) - Information Flow Control Framework for Java*. URL: https://pp.ipd.kit.edu/projects/joana/ (visited on 25/01/2018).