Hybrid Taint Analysis for Java EE

Florian D. Loch
Martin Johns
Martin Hecker
Martin Mohr
Gregor Snelting

SAP Security Research & Karlsruhe Institute of Technology

ABSTRACT

We present a new approach to protect Java EE web applications against injection attacks, which can handle large commercial systems. We first describe a novel approach to taint analysis for Java EE, which can be characterized by "strings only", "taint ranges", and "no bytecode instrumentation". We then explain how to combine this method with static analysis, based on the JOANA IFC framework. The resulting *hybrid* analysis will boost scalability and precision, while guaranteeing protection against XSS. The approach has been implemented in the *Juturna* tool; application examples and measurements are discussed.

CCS CONCEPTS

• **Security and privacy → Software security engineering**; **Web application security**;

KEYWORDS

Injection attacks, XSS, Java EE, Taint Analysis, Information Flow Control

ACM Reference Format:
Florian D. Loch, Martin Johns, Martin Hecker, Martin Mohr, and Gregor Snelting. 2020. Hybrid Taint Analysis for Java EE. In *The 35th ACM/SIGAPP Symposium on Applied Computing (SAC '20), March 30-April 3, 2020, Brno, Czech Republic.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3341105.3373887

1 INTRODUCTION

Injection vulnerabilities in software are widespread. Injection attacks are ranked first place in the *OWASP Ten Most Critical Web Security Risks* list; while cross-site scripting attacks (XSS – a particular species of injection attacks) are ranked seventh [25]. Taint tracking has been proven effective to detect a wide range of injection attacks [13, 15, 21]. But in particular for Java and Java EE applications, taint tracking has severe downsides – there is a massive run time overhead, and/or rather destructive modifications of the JVM are necessary.

In this contribution we present a new, hybrid approach to taint tracking for Java (EE). In particular, we combine new, light-weight dynamic taint tracking techniques with highly precise, static information flow control analysis. We first describe a novel approach to taint analysis for Java EE, which can be characterized by "strings only", "taint ranges", and "no bytecode instrumentation". We then explain how to combine this method with static analysis, based on the JOANA IFC framework. This results in reduced run-time overhead and also provides a light-weight integration into the Java VM. The approach was implemented in form of the *Juturna* tool. Juturna needs only minimal modifications to an application's bytecode and injects a modified standard library into the runtime but requires no JVM modifications. It provides scalability and precision for commercial applications, setting it apart from other approaches. Measurements based on the Stanford SecuriBench Micro benchmark suite demonstrate improved scalability for the Juturna approach.

1.1 Scenario and Attacker Model

Juturna was designed to mitigate injection attacks in Java in general, but for now implementation puts special emphasis on XSS vulnerabilities contained in Java EE servlets running in a production (commercial) setting. With such often only bytecode is available, but no source code, and performance might be more mission critical than a very high recall. Juturna provides support for servlet containers such as Apache Tomcat that implement the Java EE Web profile.

In a nutshell, the idea behind a reflected/non-persistent XSS attack is to make a server respond to malicious content which is provided in the request itself that then gets interpreted/executed in the security context of the browser. An attack vector could be, for example, links distributed via phishing mails containing the malicious payload as content of query parameters. A vulnerable Java Servlet, as shown in Figure 1, might now embed the content of such parameters into the page returned. With reflected XSS the browser sending the HTTP request is always the target of the attack, in case of stored/persistent XSS attacks the malicious payload usually gets included into several responses answering clean requests from several browsers.

Therefore HTTP requests can be considered the primary entry point for most injection attacks trying to exploit vulnerabilities in server-side applications like Java Servlets. In the terminology of

SAC '20, March 30-April 3, 2020, Brno, Czech Republic

ACM ISBN 978-1-4503-6866-7/20/03…$15.00
https://doi.org/10.1145/3341105.3373887

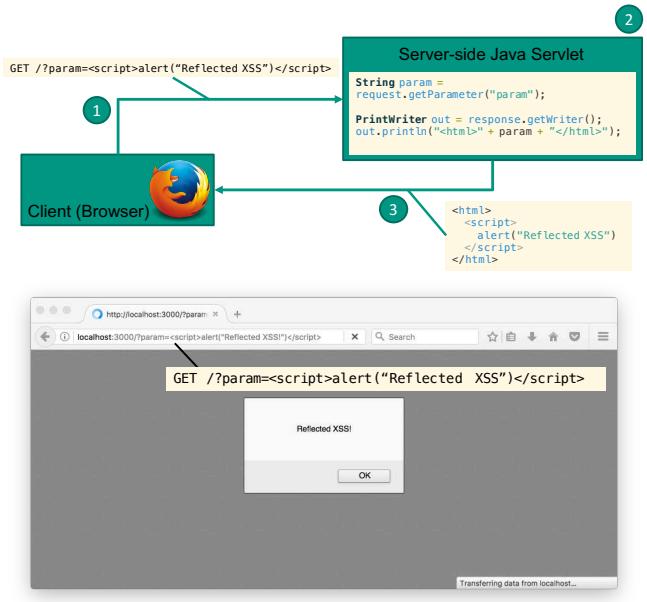

Figure 1: A simple reflected XSS attack. Top: schematic sequence of events. Bottom: exploiting this vulnerability in Mozilla Firefox.

taint analysis these entry points are called *sources*. Which parts of an application need to be considered *sinks* depends on the type of injection attack: with reflected XSS sending a response is a sink, with SQL injections it would be the respective database driver.

In the context of reflected XSS attacks we assume that all communication is already secured against any modification by TLS in order to actually require the server to embed the malicious payload into the response. But the attacker is capable of offering a starting point to potential victims, e.g., a link in a phishing email. Thus, we assume that the attacker is able to control every part of a HTTP request as specified in [8].

1.2 Taint Analysis vs. Information Flow Control

Taint analysis is a standard device against injection attacks and XSS. Taint analysis is a purely *dynamic* analysis: It tags data objects from attacker-controlled input *sources* as tainted, and dynamically checks whether (parts of) tainted data can reach sensitive *sinks*, e.g., entry points of SQL or JavaScript interpreters. When data originating from a taint source reaching a sink this is called a *taint flow* or a *taint leak*. Taint analysis usually requires either support from the runtime[1], instrumentation of bytecode or, as used in Juturna, modification of the standard library. It will then discover any explicit program leak – that is, any flow from tainted artifacts to critical execution points – without requiring a developer to have taint tracking in mind when designing an application [1, 13, 15, 21]. Taint analysis however is restricted to one specific execution path and can thus neither provide guarantees about a program in general; nor will it discover so-called implicit leaks (where not data but control flow is corrupted by an attacker). This is a disadvantage compared with static analysis mechanisms. But on the other hand, some taint trackers (such as Juturna) can handle language metaprogramming concepts like Java Reflection, dynamic class loading and similar features – which are difficult to handle for any static analysis.

Information flow control (IFC [18, 27]) guarantees to find all explicit and implicit leaks in a program, typically by performing a *static* analysis.[2] IFC will discover both violations of confidentiality (i.e. private information flows to public ports) and integrity (i.e. tainted input can influence critical computations). Technically, IFC checks a so-called noninterference property. Various noninterference criteria and analysis algorithms have been developed, which vary greatly with respect to technology (type systems vs. formal verification vs. program analysis), supported language (toy language vs. full Java or C), soundness (i.e. all leaks are guaranteed to be found), precision (i.e. low number of false alarms), scalability (i.e. big programs) and practicability (annotation overhead and tool support). IFC is more powerful than taint analysis because it guarantees to discover all potential leaks, not just explicit leaks. But the price to pay is not only a limited scalability – today, analysing 5MLoc is still out of the question – but also a sometimes high number of false alarms. IFC research tries to minimize false alarms, while maintaining soundness and improving scalability – due to decidability problems[3] one has to sacrifice soundness for precision, and vice-versa. Today, the theory of IFC is widely developed, but only very few available IFC tools (such as Andromeda [29] and JOANA [17]) can handle full Java.

Previous authors have used taint tracking to improve IFC analysis, but our approach is just the opposite: We will improve taint tracking by integrating IFC techniques.

2 THE JUTURNA APPROACH TO TAINT ANALYSIS

Juturna uses several techniques to improve precision and scalability of taint analysis. In fact, these techniques alone – even without the IFC integration – make Juturna more precise and scalable than other taint analysis solutions for Java EE. In the following, the most important techniques are sketched, while their implementation is described in section 4.

2.1 Strings Only

One fundamental design decision for Juturna was to taint-track only strings resp. characters inside strings. Previous taint trackers for Java, such as the Phosphor system [1, 2], have tracked all data types by augmenting all bytecode instructions and/or massively modifying the JVM. Thus, high overheads resulted (about 50% for Phosphor, orders of magnitude for some other tools).

In contrast, Juturna assumes that it is enough to taint-track for characters being part of strings (i.e. instances of type java.lang.St

[1]For example, basic taint tracking functionality is embedded into the official interpreters of Ruby and Perl

[2]For multi-threaded programs, modern IFC tools will also discover probabilistic leaks, which result from subtle influences of tainted values to interleaving and scheduling. In this paper we do not explore probabilistic leaks, but the Juturna approach can easily be expanded to cover multi-threaded programs.

[3]See the famous Rice Theorem

ring, java.lang.StringBuilder, java.lang.StringBuffer). Restriction to strings and ignoring plain char primitives, of course, destroys soundness as illegal flows might propagate through integers etc. But in practice string-based taints are the most relevant attacks. Restriction to strings has the huge advantage of boosting scalability – allowing to handle commercial software in performance critical environments, i.e. in productive use in data centers etc. – while still catching most injection attacks. Thus, the "strings only" approach is "sound enough" in our attacker scenario (injection attacks via HTTP) and seems to be a reasonable tradeoff.

Note that it is essential for precision to track individual characters in strings independently from each other – otherwise many false alarms will result. For efficient tracking of characters in strings, Juturna uses taint ranges.

2.2 Taint Propagation and Taint Ranges

Taint Propagation. To illustrate the subtleties of taint propagation, let us consider a few examples. Conceptually, every string object includes a tainted[] array, which for every character in the string stores its taint status. Thus, a call like c = append(a,b); must recompute the taint status as follows:

$$c.taint[i] = a.taint[i], \quad 1 \le i \le a.length() \quad (1)$$

$$c.taint[i + a.length()] = b.taint[i] \quad 1 \le i \le b.length() \quad (2)$$

and similar for other string methods. These computations are easy enough,[4] and part of the modified java.lang.String methods (see below). We will not describe them in detail here; see [22] for details.

But there are other, more conceptional issues about taint propagation. For example, equality tests do not (and must not) consider the taint array. This leads to problems with the concept of Java's string pool as it will provide just one internal copy for *internalized* strings (literals and strings on which String#intern has been called) with the same characters, completely ignoring possibly different taint arrays. Likewise, serialization ignores taint information. All these – and similar – issues can negatively affect precision and/or soundness. However it would be very costly to resolve all these issues.

For Juturna we thus decided to accept these imperfections for the following reason: Juturna is a compromise between precision, soundness, and scalability; optimized for commercial Java EE applications. In commercial settings, it is much more important to find the most common security leaks fast, with no false alarms. If soundness (i.e. a *guarantee* that all leaks are found) is required, one must resort to full IFC anyway. In practice, we recommend a two-stage approach: use Juturna on all executions and additionally use JOANA for the full analysis of selected, critical modules.

Taint Ranges. To optimize scalability and enable the differentiation between different sources of taint information and different levels as mentioned above, Juturna does not actually use a simple tainted[] array. Instead, it uses a novel approach named *taint ranges*.[5] A single taint range stores information on the taint state of a range of characters in a string. If a string has several, disconnected tainted parts – possibly originating from different taint sources – multiple taint ranges are needed. Taint ranges are, considering realistic use cases, much more efficient than taint arrays, in particular for long strings where taint originates from just one source.

A taint range consists of the following information: start and end as absolute indices, represented by two 4-byte integer values.

```
1  public class Aliasing5 {
2    private static final String FIELD_NAME = "name";
3      protected void doGet(HttpServletRequest req,
4                           HttpServletResponse resp)
5                          throws IOException {
6          StringBuffer buf = new StringBuffer("abc");
7          foo(buf, buf, resp, req);
8      }
9
10     void foo(StringBuffer buf, StringBuffer buf2,
11              ServletResponse resp, ServletRequest req)
12             throws IOException {
13       String name = req.getParameter(FIELD_NAME);
14       buf.append(name);
15       PrintWriter writer = resp.getWriter();
16       writer.println(buf2.toString());  /* BAD */
17   }
18 }
```

```
1  public class Inter10 {
2    private static final String FIELD_NAME = "name";
3    protected void doGet(HttpServletRequest req,
4                         HttpServletResponse resp)
5                         throws IOException {
6        String s1 = req.getParameter(FIELD_NAME);
7
8        String s2 = foo(s1);
9        String s3 = foo("abc");
10
11       PrintWriter writer = resp.getWriter();
12       writer.println(s2);              /* BAD */
13       writer.println(s3);              /* OK */
14   }
15   private String foo(String s1) {
16     return s1.toLowerCase().substring(0, s1.length()-1);
17   }
18 }
```

```
1  {
2    "additionalClasspaths": [
3      "lib/*"
4    ],
5    "taintSourceCategories": [
6      {
7        "name": "URL_PARAMETER",
8        "severity": "ACTUAL_SOURCE"
9      }
10   ],
11   "sources": [
12     {
13       "fqn": "I:javax.servlet.ServletRequest.
14 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣getParameter(java.lang.String)",
15       "taintSource": "URL_PARAMETER"
16     }
17   ],
18   "sinks": [
19     {
20       "fqn": "org.apache.catalina.connector.
21 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣CoyoteWriter.print(java.lang.String)",
22       "parameters": [
23         {
24           "paramIndex": 0,
25           "forbiddenSources": [
26             "*"
27           ],
28           "mitigation": "LOG"
29 }]}]}
```

Figure 2: Top: Two small examples taken from the SecuriBench Micro testsuite. Bottom: Example source/sink specification for Juturna given as JSON.

[4] There are some subtleties with Unicode, e.g., toUpperCase() applied to "ß" transforms a 1-char string into a 2-char string ("SS"). This, when not being handled, would cause a misalignment between characters and taint information in a string. Juturna is able to handle such cases correctly.

[5] Taint ranges were developed by the second author while at SAP, and improved upon in Juturna

```
1 ==========>
2 Taint incident occurred at:
3 org.apache.catalina.connector.CoyoteWriter.
4     print(CoyoteWriter.java)
5 [...]
6
7 Further information:
8 Actual sources: URL_PARAMETER
9 Forbidden sources: *
10 Mitigation strategy: LOG
11 Tainted value (up to first 300 characters; not all
12   of these characters are necessarily tainted): hoar
13 TaintInformation object: TaintInformation(
14   TaintRange(start=0, end=4, source=TaintSource(name=
15   'URL_PARAMETER', id=-32765, level=ACTUAL_SOURCE)))
16 <==========
```

```
1 ==========>
2 Taint incident occurred at:
3 org.apache.catalina.connector.CoyoteWriter.
4     print(CoyoteWriter.java)
5 [...]
6
7 Further information:
8 Actual sources: URL_PARAMETER
9 Forbidden sources: *
10 Mitigation strategy: LOG
11 Tainted value (up to first 300 characters; not all
12   of these characters are necessarily tainted): abcHoare
13 TaintInformation object: TaintInformation(
14   TaintRange(start=3, end=8, source=TaintSource(name=
15   'URL_PARAMETER', id=-32765, level=ACTUAL_SOURCE)))
16 <==========
```

Figure 3: The Juturna prototype reports on discovered leaks for examples in figure 2. Sink and source are given for a taint incident; the prototype also displays taint ranges involved.

Additionally, a 1-byte id refers to the taint source type that has been assigned to a taint source in the configuration. A taint source type itself is, as shown in line 8 in figure 2 (bottom), assigned to one of a list of predefined severity levels with different semantics. Distinguishing between different kinds of taint and enabling different behavior based on the taint source would not be possible at all with the naive, array-based approach.

The main motivation for taint ranges is the fact that taint information is usually homogeneous: a whole substring of tainted characters (or even the whole tainted string) will come from the same source. Thus, a long tainted string (of say length 2^{20}) resulting from the same source does not need a taint array of the same size, but just the additional 9 bytes. Several taint ranges can be attached to one string. As a character can, by definition, only originate from a single taint source, taint ranges can only override each other but cannot overlap - the implementation needs to check and enforce this, making taint propagation more difficult. For example, insertion of a character sequence into a string might result in a new range and possibly also in splitting up an existing one. Juturna uses a sorted list of taint ranges, and puts serious effort into a fast implementation. E.g., using absolute start and end indices allows to apply binary search for all lookup operations. More details are explained in section 4.

2.3 Specification of sources and sinks

Specific methods are declared as sources or sinks of, potentially tainted, objects. In figure 1, getParameter is a source returning tainted values and println a sink not allow to receive tainted values as one of its parameters. Figure 2 (top) shows an example from the SecuriBench Micro testsuite. Both code snippets can be abused in order to perform a reflected XSS attack because they return content potentially controlled by the attacker as part of the response. Aliasing5 uses an alias (buf2) to refer to a string becoming tainted by appending name in line 14. On a syntactical level this relationship is hard to spot, on the lower, runtime level Juturna is working on it is trivial. Inter10 calls foo twice - once receiving a clean string and once a tainted one. Both end up in the same sink but only for one an alarm should be yielded. This is again an easy catch for taint tracking systems, although some string operations actually creating new string instances are performed.

Juturna offers a configuration language based on the JSON framework, which is used to specify sources and sinks as well as additional information such as taint source categories. Figure 2 (bottom) provides the specification for Figure 2 (top): After declaring the taint source categories, just URL_PARAMETER in this case, sources and sinks get declared. A single taint source is configured, the getParameter method declared in the ServletRequest interface. As this example configuration was written with the Apache Tomcat servlet container in mind, the specific implementation (CoyoteWriter) of the otherwise too generic and broadly used PrintWriter interface used within Tomcat gets declared as sink. Due to this Juturna will add bytecode to check the taint state of the first parameter (index 0) at runtime.The examples contain leaks which are reported by Juturna – as shown in figure 3.[6] More details on source/sink specification and instrumentation will be presented in section 4. This configuration has to be provided by a security engineer analysing and assessing an application.

2.4 API Extensions and Bytecode Augmentation

Juturna does not need a modified JVM, but employs just two basic mechanisms: a) bytecode augmentation, and b) modification of standard API classes. As for the latter, the source for java.lang.string etc. has been manually modified and augmented with additional methods. In addition, standard classes such as java.lang.regexp had to be modified as well. The resulting augmented standard libraries are incorporated into the standard JVM via the command line parameter -Xbootclasspath.[7]

In addition, bytecode has to be augmented to enable taint sources to initialize the taint information in string objects and taint sinks to check for such information. For string operations, the modified class libraries will propagate taint information during execution. As a consequence, all bytecode outside source/sink methods remains unchanged – which is a huge advantage compared to bytecode instrumentation approaches, which explicitly propagate taint information by adding additional instructions. Juturna also provides a helper library that can be used to taint strings programmatically at runtime – in such a case taint propagation happens without any modifications to the bytecode.

[6] Future versions of Juturna will use a more graphical presentation of leaks in the source code.

[7] -Xbootclasspath works with the Oracle, IBM, and OpenJDK JVMs. For a commercial version, an "official" integration of the augmented libraries should be used.

These bytecode instrumentations used in Juturna are applied by exploiting another command line parameter: -javaagent, which allows to register so-called bytecode transformers (i.e. implementations of ClassFileTransformer). Transformers are able to inspect and modify the bytecode of class files loaded after the JVM has finished its bootstrapping.[8] The approach works also for encrypted class files, or with network class loaders (e.g., java.net.URLClassLoader). Thus, Juturna performs all necessary bytecode augmentations at load time, not at compile time or as a preprocessing step which avoids adding complexity to build pipelines.

Some string methods are implicitly defined as sources, e.g., AbstractStringBuilder#setCharAt might introduce a tainted character into a string. In addition, sanitization functions can be explicitly named (as they "untaint" incoming objects). But Juturna will never automatically delete taint information after sanitization (as other systems do, sometimes based on questionable heuristics, e.g., [13]). Instead, Juturna provides specific taint levels. Possible taint levels are SOURCE (indicating a tainted value from a specified source method) and SANITIZED (indicating a tainted object which was fed through a sanitization method). This allows a more detailed analysis and mitigation control. For example, strategies such as "positive tainting" [15] can easily be implemented in Juturna.

3 HYBRID IFC: JUTURNA + JOANA

3.1 JOANA

JOANA is a framework for static IFC analysis developed at KIT over the last 10 years.[9] JOANA can handle full Java with arbitrary threads, scales to ca. 250 kLOC, and guarantees to discover all explicit, implicit and probabilistic leaks [17, 28]. JOANA is based on a stack of sophisticated program analysis, such as points-to analysis, program dependence graphs, program slicing, race analysis and exception analysis [17]. JOANA minimizes false alarms through flow-sensitive, context-sensitive, and object-sensitive analysis techniques [9, 16]. JOANA requires only minimal program annotations and guarantees probabilistic noninterference [3, 5]. JOANA guarantees soundness; the soundness proof for sequential noninterference was machine-checked in Isabelle [31]. JOANA was used in realistic case studies such as e-voting software [11, 19, 23], practical application is described in [12]. JOANA also provides a programming interface for integration with other tools such as verification systems [20].

Juturna does not use all JOANA features. In fact, it only uses its integrated *program dependence graph* (PDG) and program slicing possibilities. [17] describes details on dependence graphs and slicing in JOANA. Full use of JOANA – in particular the sophisticated RLSOD check for probabilistic leaks – is only necessary if a security *guarantee* is needed – but is of course much more expensive, and possible only for selected modules up to ca. 250 kLoc.

3.2 The Hybrid Approach

Program Slicing is a fundamental program analysis technique which was explored in a major international effort since the '90s, and is also an important part of JOANA. Slicing was originally invented

```
1  linewidthclass PasswordFile {
2  private String[] names;
3                  /*P: confidential*/
4  private String[] passwords;
5                  /*P: secret*/
6  // Pre: all strings are interned
7
8  public boolean check(String user,
9      String password /*P: confidential*/) {
10     boolean match = false;
11     try {
12     for (int i=0; i<names.length; i++) {
13       if (names[i]==user
14           && passwords[i]==password) {
15
16           match = true;
17           break;
18   }}}
19   catch (NullPointerException e) {}
20   catch (IndexOutOfBoundsException e) {};
21
22   return match;  /*R: public*/
23  }
24 }
```

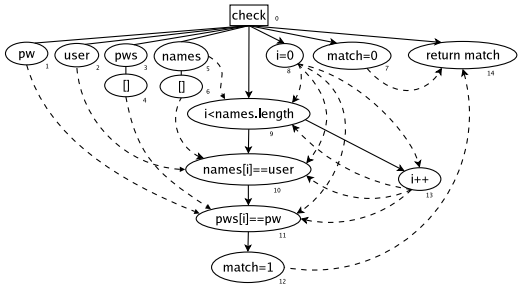

Figure 4: A simple password checker and its dependence graph (from [17]). Dashed arcs are data dependencies, solid arcs are control dependencies. Obviously there is a path from password pw **to the public return value.**

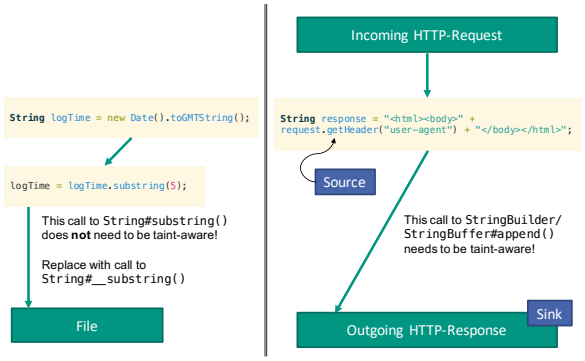

Figure 5: Example of selective taint tracking. On the left, a string not tainted and not flowing into a sink makes taint tracking superfluous. On the right, a tainted string returned from a source (...#getHeader()) gets concatenated with a literal; the code uses a StringBuilder **for concatenation. As the result ends up in a sink,** StringBuilder#append() **must be taint aware.**

3.2 The Hybrid Approach

Program Slicing is a fundamental program analysis technique which was explored in a major international effort since the '90s, and is also an important part of JOANA. Slicing was originally invented to support debugging: for a given program point, it will determine *all* statements or expressions which may influence the values computed at that point. This set of statements is called *backward slice*

[8] Transformers are thus similar to "aspects" in aspect-oriented programming. We omit technical details of the -javaagent mechanism, see the Java literature.
[9] http://joana.ipd.kit.edu provides download, webstart application, and further information.

and should, for precision, be as small as possible – but never too small. Precise slicing for full languages is absolutely nontrivial. Several hundred research papers have – over more than twenty years – eventually led to slicing algorithms for full Java, which are provably sound (i.e. all possible flow is represented in the slice, guaranteed) and very precise (i.e. slices are as small as possible) due to the following techniques (see [16, 17] and the slicing papers cited therein):

- Slicing is flow-sensitive, thus respects statement order. This massively improves precision and reduces false alarms, in particular for large programs.
- Slicing is context-sensitive, therefore it distinguishes several calls to the same method. Context-sensitivity is even more effectful than flow-sensitivity.
- Slicing is object-sensitive, that is, it distinguishes a field in different objects of the same class.
- For multi-threaded programs, slicing is thread-sensitive and will determine which statements can potentially be executed in parallel.
- Slicing uses sophisticated points-to analysis, exception analysis, and other program analysis techniques.
- Slices include explicit, implicit, and probabilistic flow of information.

Slices are computed using PDGs. To give the reader an idea of slicing figure 4 presents a Java method for password checking and its (intra-procedural) PDG. The backward slice from the return statement is obtained by following all edges from the return node backwards and includes the secret password. It thus demonstrates that information can flow from the secret password (a source) to the public result (a sink).[10]

The slicing machinery, which is part of JOANA, can contribute important information which will speed up Juturna considerably: As said, slicing can determine for any two program points whether any information can flow from the first to the second. More general, for a given sink, it can determine all program points which can influence the sink (backward slicing), and for a given source it can determine all program points which can be influenced by the source (forward slicing). For a program point p, we write $fsl(p)$ and $bsl(p)$ for forward and backward slices.

Forward slices can be exploited by Juturna as follows: Taint propagation needs to happen only along execution paths from sources to sinks. If there is no sink in the forward slice of a tainted object from some source, taint propagation is not needed for this instance. Likewise, if no source is in the backward slice of a sink, taint propagation to the sink is not needed, and the corresponding taint ranges can be deleted. Obviously, the *intersections* of forward and backward slices – known as *chops* – are the only program regions where taint propagation and taint ranges are necessary. As slices and chops are computed statically by an analysis of the bytecode, the runtime overhead for taint analysis will be reduced considerably because not the whole application needs to track taint information any longer, only the parts contained in the chop need to do so. This approach is called "hybrid taint tracking" or "selective taint tracking". In fact, for all possible pairs of sources s and sinks t in program P JOANA will compute the chop $ch(s,t) = fsl(s) \cap bsl(t)$. The chops thus determined are the program regions where taint propatation must actually take place, and are given by[11]

$$\text{unsafe}(P) = \bigcup_{\text{source } s,\ \text{sink } t\ \in PDG(P)} fsl(s) \cap bsl(t)\ .$$

An example of selective taint tracking is shown in Figure 5: on the left, taint tracking is superfluous as the string cannot flow to a sink; while on the right, taint ranges and taint propagation must be applied as the HTTP request is a taint source, and the HTTP response is a security sensitive sink. In practice, the size of unsafe(P) is usually small compared to total program size.

4 IMPLEMENTATION

Taint Ranges. Besides the taint ranges itself (9 bytes per range as described above), a container class provides methods for sorted lists of taint ranges, as well as fast insertion, lookup, appending etc. Use of binary search provides $O(\log_2 n)$ lookup and insertion times; this works only because taint ranges have absolute indices. One might think of using balanced trees of taint ranges, or other implementation improvements, but we consider this future work – the current implementation already scales well (see section 5).

Library Augmentations. Next, several augmentations have been implemented for the library classes `String`, `AbstractStringBuilder` resp. its two implementations `StringBuilder` and `String- Buffer`, `regex.Matcher`. Some augmentations have been described above, so let us just mention the modifications to regular expression matching. Functions for regular expression matching uses low-level string methods like `charAt()`, which just returns a primitive `char` to which no taint information can be attached (as we only provide this for characters being part of a string type), to construct a potential match. To still keep track of the taint information the sourrounding code had to be adjusted.

Bytecode augmentation. As mentioned before, the `-javaagent` mechanism is used to augment bytecode for methods declared to be source or sinks. Figure 6 top shows the actual implementation of class `SourceInstrumenter`, which modifies the bytecode for every source. Figure 6 bottom shows the declaration of a sink and a taint source category in JSON. Such specifications must be provided by the engineer. Juturna then adds extra bytecode to the beginning of the declared methods that will then, at runtime, check whether the actual parameter value (i.e. at least one character contained in it) is originating from a source that is contained in the list of forbidden sources/source classes.

For sinks, several checking options may be specified. The method parameter which actually corresponds to the sink must be specified (parameter 2 in line 5 of Figure 6 bottom), and illegal sources and taint classes in case a tainted object reaches the sink may be specified. In the example, the tainted string reaching the sink method must not originate from any source ("*") except for a source named "ASOURCE_THAT_IS_OK"; and the latter is allowed only if the taint category of the tainted object is "SANITIZATION_FU

[10] Indeed, every password checker reveals a small amount of secret information, namely whether the password was correct or not. The engineer must decide whether the discovered flow is relevant.

[11] Note that $fsl(s) \cap bsl(t)$ is a correct, but unprecise chop. JOANA uses the much more precise context-sensitive chops from Reps et al. [26].

```
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}")
7  ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣does not implement the TaintAware interface! Therefore it cannot
       be marked as source!""")
8          }
9
10         val stringPoolingProtection = if (method.returnType.name == "java.lang.
               String") {
11             """
12 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣$RETURN_VALUE_REFERENCE = new java.lang.String(
       $RETURN_VALUE_REFERENCE);
13 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣//$RETURN_VALUE_REFERENCE is a placeholder for $_.
14 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣//Needed because a literal $ cannot be used in multi-line
       strings.
15 ␣␣␣␣␣␣␣␣␣␣␣␣"""
16             } else {
17                 "//just needed for instances of type String"
18             }
19
20             method.insertAfter("""
21 ␣␣␣␣␣␣␣␣␣␣␣␣if ($RETURN_VALUE_REFERENCE == null) {
22 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣return null;
23 ␣␣␣␣␣␣␣␣␣␣␣␣}
24 ␣␣␣␣␣␣␣␣␣␣␣␣$stringPoolingProtection
25 ␣␣␣␣␣␣␣␣␣␣␣␣com.sap.juturna.taintHelper.THelper.get($RETURN_VALUE_REFERENCE).
       addRange(0, $RETURN_VALUE_REFERENCE.length(),
26 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣com.sap.juturna.taintStorage.TaintSource.getOrCreateInstance("${
       sourceTuple.taintSource}"));
27
28 ␣␣␣␣␣␣␣␣␣␣␣␣return $RETURN_VALUE_REFERENCE;
29 ␣␣␣␣␣␣␣␣""")
30
31             return true
32         }
33
34         /* ... */
35     }
```

```
1  {
2    "sinks": [{
3      "fqn": "java.lang.String.replaceAll(java.lang.String, java.lang.String)",
4      "parameters": [{
5        "paramIndex": 1,
6        "forbiddenSources": [
7          "*", "!A_SOURCE_THAT_IS_OK"
8        ],
9        "mitigation": "CUSTOM:com.example.MyCustomTaintCheck"
10     }]
11   }],
12   "taintSourceCategories": [{
13     "name": "A_SOURCE_THAT_IS_OK",
14     "severity": "SANITIZATION_FUNCTION"
15   }]
16 }
```

Figure 6: Top: Implementation of bytecode augmentation for methods marked as sources; Bottom: JSON specification for a sink method.

NCTION". The possible action after discovering an illegal, tainted object may also be specified; possible actions are just log the incident, throw an exception, ignore the incident, or execute specific handler code provided by the engineer.

For sources, Juturna adds bytecode to all possible return paths of the stated methods that attaches the taint information to the returned values at runtime. This code calls THelper in line 25 to initialize the taint range for the string object which is created/returned by the source method. Lines 11-17 prevent to taint strings which are actually pooled (in the constant pool): The string object (but not its char[] array!) is copied and a new TaintInformation instance (the container class for taint ranges) initialized. This trick

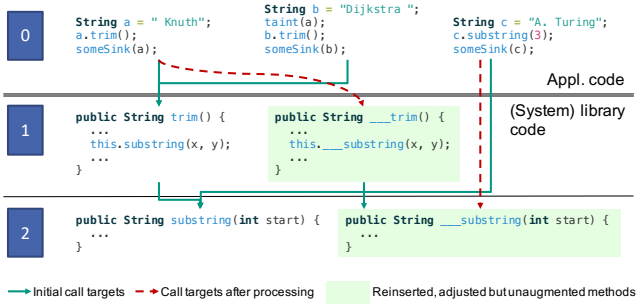

Figure 7: Replacement of augmented string functions by unaugmented ones, due to selective tainting.

prevents that taint information propagates irregularly through the constant pool in case the value returned from a method has been interned, i.e. put into the constant pool.

This process relies on the *Javassist* tool for bytecode manipulation. Let us finally mention that in some cases, method implementations of CharSequence, such as setChar() and replace(), must be augmented to initialize taint ranges (similar to explicit sources); we omit details due to lack of space.

JOANA interface. The JOANA adapter receives a class file containing the entry point (main()) of the software to be analysed, and determines the sinks which are in the forward slice of tainted sources (according to the taint specification, see above). This list of safe sinks is exported in JSON format to the Juturna configurator. The configurator will *not* augment the code of sinks if these are outside the slice, thus benign. This implies that for all String etc methods, the original code must be available as well as the augmented code. Hence all the String etc methods are duplicated, and unique method names provided (e.g., ___ prefixes). The slice-based choice between augmented and non-augmented method implementations must of course be recursively applied to all auxiliary methods called. Technically, the choice from augmented back to unaugmented calls is implemented by cloning the call target object with an object from the unaugmented class. It is important that the transitive process of replacing augmented methods by unaugmented ones, and cloning call target objects accordingly, is done in a consistent manner, and respects library interfaces. Figure 7 shows an example of some strings which are operated on with string methods, and how these methods are replaced by their unaugmented versions, because JOANA determined that the call parameters cannot be tainted.

JOANA is run in a specific Juturna configuration. Implicit flows are disabled, as well as the check for probabilistic noninterferences. Such configurations may be selected using JOANA options or command line parameters (cmp. [12]).[12] JOANA also offers a programming interface where starting points for slices resp. chops can be selected, and the actual slices or chops be computed and later examined. Figure 8 shows the use of this interface by Juturna; due to lack of space we omit details. We would like to add that Juturna

[12]Note that in JOANA "standalone" or "IFC" mode, implicit and/or probabilistic leaks are enabled; thus soundness is guaranteed (no missed leaks), but precision and scalability decrease (more false alarms, higher analysis time).

```
1  fun analyze(sdg: SDG, sourceFQNs: List<String>, sinkFQNs: List<String>): List<
       SDGNode> {
2      // "data class" is shorthand in Kotlin for declaring a record-like structure
3      data class CallNodeInfo(val callNode: SDGNode, val thisParameter: SDGNode,
               val thisParameterM
4  emberNodes: LinkedHashSet<SDGNode>)
5
6      val entryNodes: List<SDGNode> = getEntryNodes(sdg)
7      val sourceEntryNodes: List<SDGNode> = filterByName(entryNodes, sourceFQNs)
8      val sinkEntryNodes: List<SDGNode> = filterByName(entryNodes, sinkFQNs)
9      val sourceNodes: List<SDGNode> = getAffiliatedExitNodes(sourceEntryNodes)
10     val sinkNodes: List<SDGNode> = getAffiliatedFormalInNodes(sinkEntryNodes)
11     val rawCallNodes: List<SDGNode> = getAllCallNodes(sdg)
12     val processedCallNodes: List<CallNodeInfo> = processCallNodes(rawCallNodes)
13     val slicer = SummarySlicerForward(sdg)
14
15     var changed = false
16     do {
17         changed = false
18
19         val nodesInForwardSlice: Collection<SDGNode> = slicer.slice(sourceNodes)
20
21         for (item in processedCallNodes) {
22             if (item.thisParameter in nodesInForwardSlice) {
23                 continue
24             }
25
26             if (isAny__thisParameterMemberNode__containedInSlice(item,
                       nodesInForwardSlice)) {
27                 sourceNodes.add(item.thisParameter)
28                 changed = true
29             }
30         }
31     } while (changed)
32
33     val chopper = NonSameLevelChopper(sdg)
34     val nodesInChop = chopper.chop(sourceNodes, sinkNodes)
35     val benignCallNodes = ArrayList<SDGNode>()
36
37     for (item in processedCallNodes) {
38       if (item.thisParam !in nodesInChop) {
39         benignCallNodes.add(getNodeOfTargetOfCall(sdg, item.callNode))
40       }
41     }
42
43     return benignCallNodes
44 }
```

Figure 8: Code selecting benign sinks by calling JOANA interface methods for an SDG. Sources and sinks are given as unique function identifier lists to the analyse() **method.**

can always be used without JOANA, but of course will then not profit from selective tainting.

5 MEASUREMENTS

In this section we present results of a performance evaluation for Juturna. Note that these are preliminary measurements; a systematic evaluation of soundness, precision, scalability, and comparison with competing tools has just begun. Still, the preliminary performance data confirm our design decisions for Juturna.

The performance evaluation is based on the Stanford Securibench Micro benchmark; which provides a set of ca. 100 (basic to complex) Java EE example servlets, many of them with XSS vulnerabilities. Apache Tomcat has been used as the underlying Java EE implementation. All measurements used Oracle JRE 1.8.0_111, running on a MacBook Pro with 2.2.GHz, under the *Java Microbenchmarking Harness* tool.

Effect of taint ranges. Figure 9 shows memory requirements for naive tainted[] arrays vs. taint ranges. It also shows the memory consumption of the corresponding original string objects. The naive approach always generates memory consumption which is

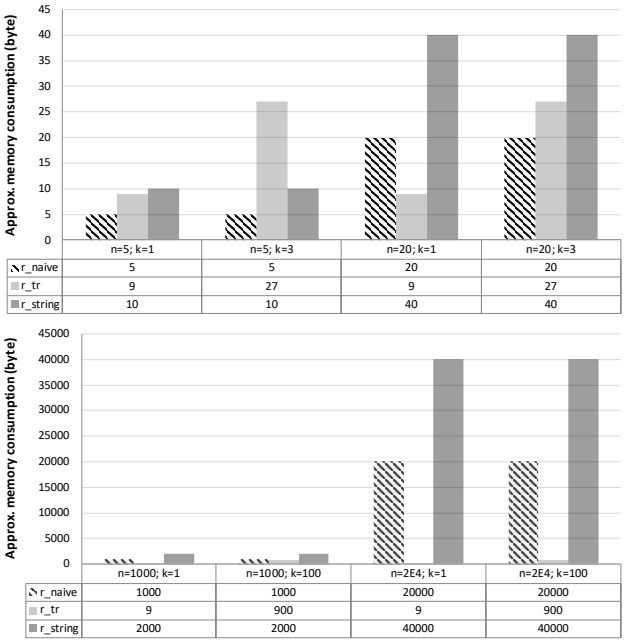

Figure 9: Memory requirements of naive taint[] **array vs. taint ranges. Shaded: unchanged API (no taint info). Grey: taint ranges. Dark: naive** tainted[] **array.**

proportional to the memory consumption of the original strings. The figure shows that the proportion factor is ca. 50% (shaded and dark columns; n is the string length). For taint ranges (grey columns), the memory consumption depends strongly on the number k of taint ranges and on the string length. The figure shows that for short strings ($n \leq 20$) taint ranges are not really better than the naive approach (for $k = 1$) or even worse (for $k = 3$). For long strings ($n = 10^3$ to 10^4) however the memory savings by taint ranges is huge (even for $k = 100$). This behaviour comes from the fact that taint ranges come with an initial overhead (11 byte per taint range), which however amortizes quickly as n is growing. Note that in today's commercial systems, k tends to be very small (near 1), while n tends to be large.

Next, the Juturna tainting overhead is determined. Various benchmarks of string functions were used, and the throughput (average number of string operations per second) was determined. Figures 10 and 11 present two typical measurement results, where the benchmark for Figure 11 contains more string operations. The first bar displays throughput for the standard JVM+API. The other 5 bars display throughput of the standard JVM plus augmented string API, for string length $n = 250$ (left) as well as $n = 10^4$ (right); where the number of taint ranges varies between $k = 0$ and $k = 100$. First note that for $k = 0$ (i.e. no tainting) Juturna has no overhead! For $n = 250$ throughput is however ca. 50% for $k = 1$, and will degrade to ca. 30% for $k = 10$. For $n = 10^4$, throughput is generally much smaller (as strings are much longer), but does not degrade as much for higher k. In figure 11, throughput degration for higher k is worse than in figure 10, but the general behaviour is similar.

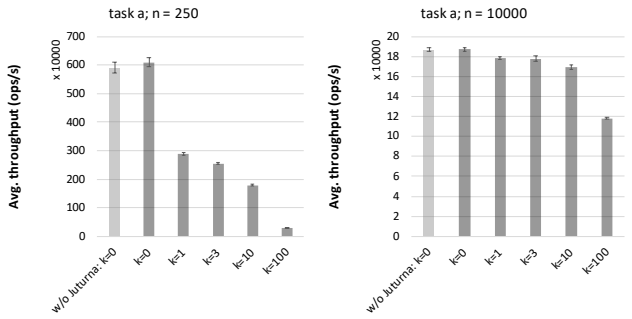

Figure 10: Throughput of string operations for standard string API vs. the augmented Juturna string API; benchmark]"Task a".

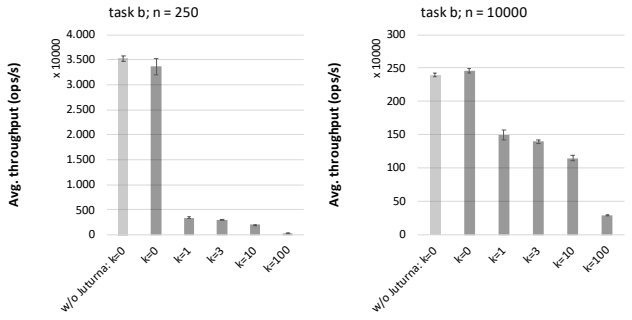

Figure 11: Throughput of string operations for standard string API vs. the augmented Juturna string API; benchmark "Task b".

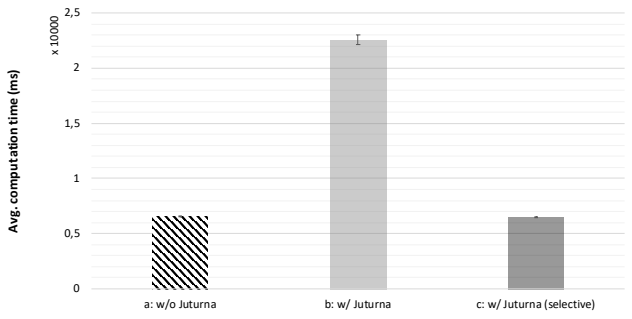

Figure 12: Effect of selective tainting: a) standard JVM/API without taint analysis, b) Juturna, c) Juturna + JOANA.

These results imply that Juturna has – in contrast to taint trackers based on bytecode instrumentation – almost no overhead for untainted or mildly tainted programs. This desirable feature is a result of the "strings only" approach, which allows to get rid of instrumented bytecode, and replace them by source/sink/API augmentation.

Effect of JOANA.. Let us now measure the effect of selective tainting. Figure 12 shows the average computation time for a small Java SE application. In this example, Juturna taint analysis is about 4 times as slow as the standard JVM/API (due to a specific n/k ratio). Adding JOANA and using selective tainting, this overhead disappears completely! The example clearly indicates that hybrid taint analysis is worth the effort: the static JOANA analysis (which must be done once for a system resp. its critical components) is amortized by a massive reduction of runtime overhead for taint analysis. Earlier, more comprehensive evaluations of JOANA have already demonstrated that source/sink chops are usually much smaller than the full code size [10, 16]; hence the example can be considered typical. More measurements of selective tainting can be found in [22].

6 FUTURE WORK

An obvious issue in the near future is a more systematic evaluation of precision and scalability, in particular for selective tainting. We plan a large-scale evaluation based on the *OWASP WebGoat*.

As taint ranges may cause a substantial overhead for some n/k ratios, one might think of better implementations for taint ranges. One might try tree-like structures instead of sorted lists of taint ranges (but this will again increase the overhead). One more promising approach is *lazy evaluation* of taint ranges. That is, taint operations as specified in equation 1 will not be executed immediately, but are delayed, and performed only when sinks are reached. This concept of lazy evaluation is well-known in functional programming, but has also successfully been used in program analysis (e.g., as done with Andromeda [29]).

One serious issue is the use of reflection and dynamic class loading. This is popular in Java EE applications (e.g., "hot swaps"), but is very difficult to handle for any static analysis (including JOANA). Heuristics for handling reflection and dynamic class loading have been published (e.g., [4]). But in general, selective tainting cannot be used as soon as classes are loaded dynamically, and are on source/sink paths – a serious loss of soundness would otherwise result. Still, we expect most code parts susceptible to selective tainting to be static, hence the overhead reduction via JOANA should still be substantial.

7 RELATED WORK AND CONCLUSION

Many systems for Java taint analysis have been proposed, among them [1, 7, 13, 14, 21]. In particular, "strings only" was already proposed e.g. in [6]. Compared to these earlier approaches, Juturna is unique in its combination of a) "strings only", b) taint ranges, c) code augmentation only for sources and sinks, d) load-time, not link-time augmentation. This combination explains Juturna's better precision (reduced false alarms), scalability (commercial programs), and practicability (all bytecode remains unchanged until loading).

There are also static taint analysers which can handle full Java: TAJ [30] is based on thin slicing, while the successor Andromeda [29] uses demand-driven "lazy" analysis instead of PDG construction. Compared to full IFC resp. JOANA, TAJ and Andromeda omit implicit and probabilistic leaks, which considerably improves scalability, but loses soundness in the strong IFC (noninterference) sense. Of course, any full IFC tool such as JOANA can be used to perform (or improve) taint analysis; this idea is not new (see e.g., [24, 32]).

But Juturna is the first taint analysis which can handle commercial Java EE software, does not need a special JVM, and integrates IFC technology. By design, Juturna is a careful compromise between precision and soundness, optimized for commercial Java EE applications. Juturna may additionally exploit the sophisticated static IFC analysis provided by JOANA, which boosts runtime performance. Juturna thus demonstrates that careful engineering of static and dynamic program analysis can reconcile the conflict between soundness, precision and scalability in software security analysis.

Acknowledgements. This work was partially supported by BMBF in the scope of the software security competence center KASTEL. Simon Bischof provided valueable comments.

REFERENCES

[1] Jonathan Bell and Gail Kaiser. 2015. Dynamic Taint Tracking for Java with Phosphor (Demo). In *Proc. ISSTA*. 409–413.

[2] Jonathan Bell and Gail E. Kaiser. 2014. Phosphor: illuminating dynamic data flow in commodity jvms. In *Proc. OOPSLA*. 83–101.

[3] Simon Bischof, Joachim Breitner, Jürgen Graf, Martin Hecker, Martin Mohr, and Gregor Snelting. 2018. Low-Deterministic Security For Low-Deterministic Programs. *Journal of Computer Security* 26 (2018), 335–366.

[4] Eric Bodden, Andreas Sewe, Jan Sinschek, Hela Oueslati, and Mira Mezini. 2011. Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders. In *Proc. ICSE*. 241–250.

[5] Joachim Breitner, Jürgen Graf, Martin Hecker, Martin Mohr, and Gregor Snelting. 2016. On Improvements Of Low-Deterministic Security. In *Proc. Principles of Security and Trust (POST)*. Springer Berlin Heidelberg, 68–88.

[6] Erika Chin and David Wagner. 2009. Efficient Character-level Taint Tracking for Java. In *Proc. ACM Workshop on Secure Web Services (SWS '09)*. 3–12.

[7] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick D. McDaniel, and Anmol Sheth. 2014. TaintDroid: An Information Flow Tracking System for Real-time Privacy Monitoring on Smartphones. *Commun. ACM* 57, 3 (2014), 99–106.

[8] R. Fielding and J. Reschke. 2014. *Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content*. RFC 7231. http://www.rfc-editor.org/rfc/rfc7231.txt

[9] Dennis Giffhorn and Gregor Snelting. 2015. A New Algorithm For Low-Deterministic Security. *International Journal of Information Security* 14, 3 (April 2015), 263–287.

[10] Jürgen Graf. 2016. *Information Flow Control with System Dependence Graphs — Improving Modularity, Scalability and Precision for Object Oriented Languages*. Ph.D. Dissertation. Karlsruher Institut für Technologie, Fakultät für Informatik.

[11] Jürgen Graf, Martin Hecker, Martin Mohr, and Gregor Snelting. 2015. Checking Applications using Security APIs with JOANA. 8th International Workshop on Analysis of Security APIs.

[12] Jürgen Graf, Martin Hecker, Martin Mohr, and Gregor Snelting. 2016. Tool Demonstration: JOANA. In *Proc. Principles of Security and Trust (POST 2016) (Lecture Notes in Computer Science)*, Vol. 9635. Springer Berlin Heidelberg, 89–93.

[13] Vivek Haldar, Deepak Chandra, and Michael Franz. 2005. Dynamic taint propagation for Java. In *Proc. Annual Computer Security Applications Conference, ACSAC*. 303–311.

[14] William G. J. Halfond and Alessandro Orso. 2005. AMNESIA: Analysis and Monitoring for NEutralizing SQL-injection Attacks. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering (ASE '05)*. ACM, New York, NY, USA, 174–183. https://doi.org/10.1145/1101908.1101935

[15] William G J Halfond, Alessandro Orso, and Panagiotis Manolios. 2006. 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*. 175.

[16] Christian Hammer. 2010. Experiences with PDG-based IFC. In *Proc. ESSoS'10*. 44–60.

[17] Christian Hammer and Gregor Snelting. 2009. Flow-Sensitive, Context-Sensitive, and Object-sensitive Information Flow Control Based on Program Dependence Graphs. *International Journal of Information Security* 8, 6 (December 2009), 399–422.

[18] Daniel Hedin and Andrei Sabelfeld. 2011. A Perspective on Information-Flow Control. *Proceedings of the 2011 Marktoberdorf Summer School* (2011). http://www.cse.chalmers.se/{~}andrei/mod11.pdf

[19] Ralf Küsters, Enrico Scapin, Tomasz Truderung, and Jürgen Graf. 2014. Extending and Applying a Framework for the Cryptographic Verification of Java Programs. In *Proc. POST (LNCS 8424)*. Springer, 220–239.

[20] Ralf Küsters, Tomasz Truderung, Bernhard Beckert, Daniel Bruns, Michael Kirsten, and Martin Mohr. 2015. A Hybrid Approach for Proving Noninterference of Java Programs. In *IEEE 28th Computer Security Foundations Symposium (CSF)*. 305–319.

[21] Sebastian Lekies, Ben Stock, and Martin Johns. 2013. 25 Million Flows Later: Large-Scale Detection of DOM-based XSS. In *Proc. ACM SIGSAC conference on Computer & communications security*. ACM, 1193–1204.

[22] Florian Loch. 2018. Juturna: Lightweight, Pluggable and Selective Taint Tracking for Java. Master's Thesis, KIT, Fakultät für Informatik.

[23] Martin Mohr, Jürgen Graf, and Martin Hecker. 2015. JoDroid: Adding Android Support to a Static Information Flow Control Tool. In *Proc. Software Engineering (CEUR Workshop Proceedings)*, Vol. 1337. 140–145.

[24] M. Mongiovì, G. Giannone, A. Fornaia, G. Pappalardo, and E. Tramontana. 2015. Combining static and dynamic data flow analysis : a hybrid approach for detecting data leaks in Java applications. In *Proc. 30th ACM Symposium on Applied Computing*. 1573–1579.

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

[26] Thomas Reps and Genevieve Rosay. 1995. Precise Interprocedural Chopping. In *Proc. ACM SIGSOFT Symposium on Foundations of Software Engineering*. 41–52.

[27] A. Sabelfeld and A. Myers. 2003. Language-Based Information-Flow Security. *IEEE Journal on Selected Areas in Communications* 21, 1 (January 2003), 5–19.

[28] Gregor Snelting, Dennis Giffhorn, Jürgen Graf, Christian Hammer, Martin Hecker, Martin Mohr, and Daniel Wasserrab. 2014. Checking Probabilistic Noninterference Using JOANA. *it – Information Technology* 56 (Nov. 2014), 280–287.

[29] Omer Tripp, Marco Pistoia, Patrick Cousot, Radhia Cousot, and Salvatore Guarnieri. 2013. ANDROMEDA: Accurate and Scalable Security Analysis of Web Applications. In *Proc. International Conference on Fundamental Approaches to Software Engineering (FASE'13)*. 210–225.

[30] Omer Tripp, Marco Pistoia, Stephen J. Fink, Manu Sridharan, and Omri Weisman. 2009. TAJ: effective taint analysis of web applications. In *Proc. PLDI*. 87–97.

[31] Daniel Wasserrab, Denis Lohner, and Gregor Snelting. 2009. On PDG-Based Noninterference and its Modular Proof. In *Proc. PLAS '09*. ACM.

[32] Jingling Zhao, Junxin Qi, Liang Zhou, and Baojiang Cui. 2016. Dynamic taint tracking of web application based on static code analysis. *Proc. 10th International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing (IMIS)* (2016), 96–101.