BackFlow: Backward Context-sensitive Flow Reconstruction of Taint Analysis Results

Pietro Ferrara[1], Luca Olivieri[1,2], and Fausto Spoto[2]

[1] JuliaSoft SRL, Verona, Italy
{pietro.ferrara,luca.olivieri}@juliasoft.com
[2] Università di Verona, Italy
fausto.spoto@univr.it

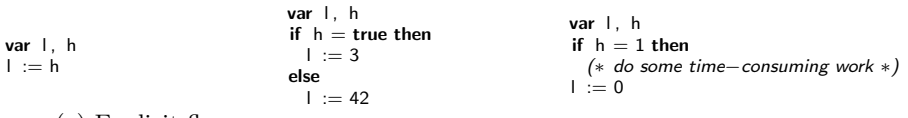
Abstract. Taint analysis detects if data coming from a source, such as user input, flows into a sink, such as an SQL query, unsanitized (not properly escaped). Both static and dynamic taint analyses have been widely applied to detect injection vulnerabilities in real world software. A main drawback of static analysis is that it could produce false alarms. In addition, it is extremely time-consuming to manually explain the flow of tainted data from the results of the analysis, to understand why a specific warning was raised. This paper formalizes BackFlow, a context-sensitive taint flow reconstructor that, starting from the results of a taint-analysis engine, reconstructs how tainted data flows inside the program and builds paths connecting sources to sinks. BackFlow has been implemented on Julia's static taint analysis. Experimental results on a set of standard benchmarks show that, when BackFlow produces a taint graph for an injection warning, then there is empirical evidence that such warning is a true alarm. Moreover BackFlow scales to real world programs.

1 Introduction

Software security vulnerabilities allow an attacker to perform unauthorized actions. In the last decade, hackers have widely exploited such vulnerabilities, in particular SQL injections and cross-site scripting (XSS), causing relevant damages. For instance, the Equifax data breach[3] relied on a command injection vulnerability. Hackers exploited this flaw to access data of hundreds of millions of Equifax customers, heavily impacting Equifax business and market value. Therefore, it is industrially relevant to *detect* and *prevent* such flaws and attacks.

Detection of cyber-attacks has been mostly based on run-time environments that monitor the system, in production, to discover anomalous situations. In this way, one discovers attacks based on the exploitation of software vulnerabilities, but also other types of attacks, such as denial-of-service through botnets. For instance, the Mirai malware[4] exploited IoT devices with default credentials

[3] https://en.wikipedia.org/wiki/Equifax#May%E2%80%93July_2017_data_breach
[4] https://techcrunch.com/2016/10/10/hackers-release-source-code-for-a-powerful-ddos-app-called-mirai/



(a) Explicit flow (b) Implicit flow (c) Side channel

Fig. 1: Different types of flows (from `https://en.wikipedia.org/wiki/Information_flow_(information_theory)`).

to gain administrative access to the device. The prevention of cyber-attacks has been also based on system safeguards, such as firewalls that block Internet traffic from and to malicious IP addresses; or on antivirus that detect and block malicious software running inside the system, that could be exploited by a hacker; or on the prevention of software vulnerabilities. In the latter case, two distinct approaches exist: *dynamic* analysis (notably, penetration testing [2]) runs the software as much pervasively as possible, in order to expose such vulnerabilities during the execution; *static* analysis, instead, builds a model of the program and detects patterns that might lead to security vulnerabilities.

Dynamic analysis does not produce false alarms: all reported vulnerabilities are real. However, it usually achieves limited coverage, since it cannot activate all possible run-time values. Static analysis, instead, can achieve high coverage, but at the price of precision. Namely, it might report false alarms that are not real flaws, since it must apply some forms of approximation to ensure the finiteness of the analysis. There exist two families of static analyzers: those based on *syntactic* reasoning, that operate locally on the abstract syntax tree of a code unit; and those based on *semantic* reasoning, that apply formal methods to approximate the overall structure of the code under analysis. The former might miss many real vulnerabilities and/or produce many false alarms, but typically scale to software of industrial size (between 100KLOCs and 1MLOCs); the latter can achieve full coverage, but are often slow or have limited precision.

Since security vulnerabilities, such as SQL injections and XSS, have heavy impact on a software system, semantic static analyzers attracted industrial attention. In particular, catching and fixing all possible software flaws before deployment is extremely valuable. Hence, the research community focused on these issues. In particular, information flow analyses [14,33] tackled the problem of tracking flows of information through a program. The problem was formalized as the detection of private information flowing into public channels. With the help of Figure 1, where `h` and `l` represent secret and public variables, respectively, one can identify three main types of flows: (i) direct flows, when a secret variable is directly assigned to a public one (Figure 1a); (ii) indirect flows, when the assignment of some public variable is performed in a branch of code whose execution is conditional on the value of a secret variable (Figure 1b); and (iii) side channels, where some observable property of the execution depends on the value of some secret variable (Figure 1c).

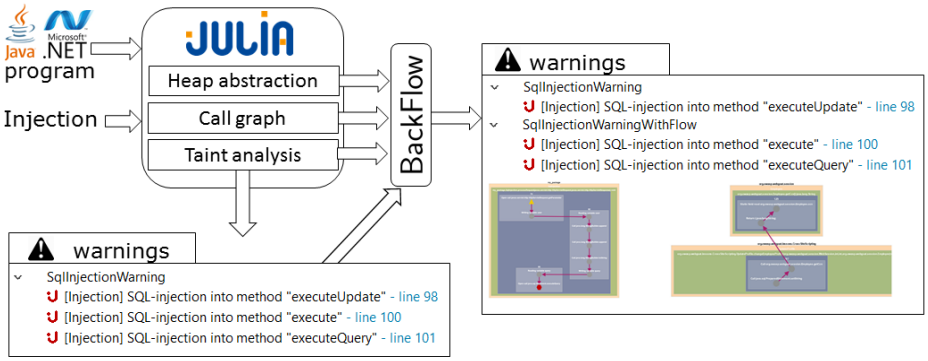


Fig. 2: The architecture of BackFlow.

All three types of flows are theoretically dangerous, since they could induce XSS, SQL injections (see for instance Fig. 7 of [7]) or leakages of sensitive data. However, in practice, hackers have rarely exploited implicit flows and side channels; moreover, their identification leads to very conservative results (too many false alarms). Therefore, *taint* analysis [3,8,28] focuses on explicit flows only, ignoring implicit flows and side channels. It has been applied to industrial software, both in dynamic and static flavor [36,41]. It checks if data coming from an untrusted source (such as user input or servlet parameters) flows, explicitly, into a trusted sink (such as the execution of an SQL query), unsanitized (that is, not correctly escaped). This analysis approximates values as Booleans (tainted/untainted), which helps scalability, since it does not require much memory and computational resources. However, it abstracts away precise information about the *exact source* of tainted data, and its *flow*.

1.1 Contribution

This paper introduces BackFlow, a new tool that reconstructs the flow of tainted data from sinks, backwards towards the sources, by building a complete *taint graph* from sources to sinks. BackFlow relies on a preliminary static taint analysis and on two standard semantic static analyses for heap abstraction and for call-graph construction. It builds a flow from a source of tainted data to a sink that receives such data. It does so in a *context-sensitive* way, that is, it considers under which circumstances the flow might exist, and discards unfeasible flows, under the information inferred by the semantic static analysis.

BackFlow has been implemented inside the Julia static analyzer [35], a semantic static analyzer for Java and C# programs, that already implements a static taint analysis [36], a call-graph construction and various heap abstractions. Figure 2 reports its overall architecture. Starting from a Java or .NET program, it first runs Julia's taint analysis. Such analysis requires to compute various heap abstractions (and in particular a creation point and a must aliasing analyses), and a call graph. The result is a set of (injection) warnings, for

potentially vulnerable program points, with the indication of the sink parameter that is reached by tainted data. However, these warnings do not specify any explicative flow about the origin of the tainted data from a source. That flow is exactly what BackFlow tries to reconstruct. If successful, it replaces the original warning with a new warning that includes a *taint graph*, representing the previously missing flow. Otherwise, the original warning is left, unchanged.

This paper discusses two experiments with BackFlow. A first, *qualitative* experiment is the analysis of WebGoat, an application developed by OWASP to teach security vulnerabilities (thus containing several kinds of explicit injections). BackFlow produces a taint graph for all true positives, while it fails for all false pasitives. Therefore, the inference of a taint graph is an effective indicator of a true alarm[5]. A second, *quantitative* experiment is the analysis of a dozen standard benchmarks. BackFlow scales to these real world applications, by running in about a fifth of the time spent for the preliminary static taint analysis. The percentage of warnings for which BackFlow reconstructs a taint graph is smaller but comparable with that for WebGoat.

This paper is structured as follows. The rest of this section introduces a running example. Section 2 discusses related work and compares it to BackFlow. Section 3 formalizes BackFlow's approach. Section 4 describes the integration of BackFlow with Julia. Section 5 describes the implementation. Section 6 reports the experiments. Section 7 concludes.

1.2 Running Example

Figure 3 reports a minimal example that needs context-sensitive information in order to infer the correct flow of tainted data from a source to a sink. In this example, method `source` is assumed to yield tainted data that method `sink` should not receive. A sound static taint analyzer must issue a warning at line 11, since tainted data actually flows into `sink` there. A backward reconstructor then would look, from line 11, for assignments to `w.f`: it goes back to line 4 and then further backwards to the beginning of method `set` (line 3). It then queries the call graph to know where `set` might be called. This occurs at both lines 10 and 16, in both cases with tainted data. However, a context-sensitive reconstructor should be able to discard the flow from line 16, since the receiver of the call to `set`, there, cannot be alias to `w` at line 11. Therefore, the only possible flow starts at line 10, goes to the field assignment at

```
class Wrapper {                                  1
  String f;                                      2
  void set(String f) {                           3
    this.f = f;                                  4
  }                                              5
}                                                6
class Bugged {                                   7
  void vulnerable() {                            8
    Wrapper w = new Wrapper();                   9
    w.set(source());                             10
    sink(w.f);                                   11
  }                                              12
}                                                13
class NotBugged {                                14
  void noise() {                                 15
    new Wrapper().set(source());                 16
  }                                              17
}                                                18
```

Fig. 3: Running Example.

[5] Note that this is an empirical result, since theoretically BackFlow might produce taint graphs for false alarms, and fail to produce taint graphs for true alarms.

line 4, continues with the field read at line 11, and ends with the sink call at the same line.

2 Related Work

Section 1 has already discussed different types of information flow analysis and taint analysis. This section discusses and compares to BackFlow the most representative static and dynamic analyses of programs, to detect security vulnerabilities and privacy leaks.

Information flow analysis has been widely applied to several programming languages. In the object-oriented context, JFlow [27] is possibly the most notable example. It is an extension of the Java programming language, that allows developers to add information flow annotations to the code, to classify variables into private or public. Then it statically checks such annotations, to discover if the value of a private variable can ever flow into a public variable. JFlow checks both implicit and explicit flows. SAILS [45] instead combines a generic information leakage analysis inside a generic static analyzer (Sample [10,17,18]) and applies this analysis to some Java benchmarks.

During the last decade, dynamic taint analysis has been widely exploited to detect, at run time, various types of security vulnerabilities and privacy leaks: the run-time environment gets augmented for tracking a Boolean mark for tainted variables, with low overhead. This augmentation can also be used to issue an alarm when a tainted value reaches a sink. Namely, TaintCheck [28] performs binary instrumentation to track tainted variables and detect vulnerabilities. Also Dytan [8] provides a generic framework for dynamic taint analysis and instantiates it to x86 executables. The dynamic taint analyzer Panorama [44] introduces the concept of *taint graphs*. For Panorama, these are very abstract, since they track how different executables propagate tainted data, but not through which exact statements. Comet [25] produces, instead, more concrete taint graphs, that represent how values in the heap or stack are affected by tainted data inside a single run-time state. BackFlow produces completely different taint graphs, that precisely represent how program statements propagate tainted data from a source to a sink.

Static taint analysis has been applied to different contexts and scenarios. During the last decade, web applications have been their most popular target, since security issues can have major impact on them. Pixy [24] applies a data flow analysis that detects XSS vulnerabilities in PHP code. Wasserman and Su [43] define a precise static taint analysis to detect SQL-injections, while JSA [39] uses dynamic information about the run-time context of a web application to reduce the rate of false alarms produced by static taint analysis. TAJ [41] applies an ad-hoc forward slicing technique to propagate tainted data. This tool produces a sort of taint graphs, but ignores dependencies through the heap, being based on the no-heap system dependence graph by Reps *et al.* [32].

There are also several tools that, starting from the results (that is, warnings) of a static analyzer, try to produce a witness through dynamic analysis or

classify automatically true and false alarms. For instance, Check 'n' Crash [13] starts from the static checks performed by ESC/Java, builts a set of constraints that must hold to produce the error reported by a warning, and then produces concrete test cases to expose the error. Aletheia [40] instead applies statistical learning to discern between true and false alarms produced by a commercial static security analyzer for JavaScript.

More recently, static and dynamic taint analyses have been applied to Android applications. FlowDroid [3] is a precise static taint analysis that detects sensitive data leaks. TaintDroid [16] performs an efficient dynamic taint analysis. MorphDroid [20] specializes the analysis for specific categories of sensitive data, for more precise and detailed results. Other approaches [5,9] introduce various extensions of information flow and taint analysis to track which kind of sensitive data is managed and potentially disclosed by Android applications.

Some works combine dynamic and static taint analysis: Saner [4] detects faulty sanitization procedures in PHP programs, while Vogt *et al.* [42] detect XSS vulnerabilities in the client browser of a web application.

As far as we know, BackFlow is the first tool that, starting from a generic static taint analysis and exploiting some supporting heap analyses, builds a context-sensitive taint graph that provides evidence of the flows of tainted data from a source to a sink.

3 Formalization

This section formalizes our approach over a simple object-oriented language.

3.1 Language

Let Programs and Classes represent the set of all possible programs and classes, respectively. An object-oriented program is an element $\mathtt{p} \in \mathsf{Programs} = \wp(\mathsf{Classes})$. A class $\mathtt{c}$ is composed of a set of fields and methods: $\mathtt{c} \in \mathsf{Classes} = \wp(\mathsf{Fields}) \times \wp(\mathsf{Methods})$, where Fields and Methods are the set of all possible fields and methods, respectively. A field $\mathtt{f}$ is a pair of its name and its type: $\mathtt{f} \in \mathsf{Fields} = \mathsf{Names} \times \mathsf{Types}$, where Names and Types are the set of all names and types (including classes in Classes and native types such as `int` or `double`), respectively. A method $\mathtt{m}$ is a pair of a list of n parameters $\mathsf{Parameters} = (\mathsf{Names} \times \mathsf{Types})^n$ and a body, represented as a control flow graph $\mathtt{cfg} \in \mathsf{CFGs}$ of basic statements: $\mathsf{Methods} = \mathsf{Parameters} \times \mathsf{CFGs}$. Control flow graphs (CFGs) are directed graphs whose vertexes are statements (formally, graphs are elements of $(\mathsf{Statements}, \mathsf{Statements} \times \mathsf{Statements})$). We assume that statements in Statements includes the label of the statement (thus we can have the same statement at different program points), but we omit it in the formalization for the sake of simplicity. The set $\mathsf{VarAtStatement} = \mathsf{Statements} \times \mathsf{Names}$ allows one to refer to a variable at a program point. Namely, $(\mathtt{s}, \mathtt{v}) \in \mathsf{VarAtStatement}$ represents variable $\mathtt{v}$ at the exit state of $\mathtt{s}$.

For the sake of simplicity, our formalization focuses on a minimal object-oriented programming language, whose statements $\mathsf{Statements}$ are either (i) the beginning of a method ($\texttt{start}$), (ii) a field read ($\texttt{y} = \texttt{x.f}$), (iii) a field write ($\texttt{y.f} = \texttt{x}$), (iv) a method call ($\texttt{y} = \texttt{x.m}(\texttt{z}_1, \cdots, \texttt{z}_\texttt{n})$), or (v) a return ($\texttt{ret x}$). Function $\mathsf{pred} : \mathsf{Statements} \to \wp(\mathsf{Statements})$ yields all predecessors of a given statement. Each statement belongs to a method. Therefore, function $\mathsf{getMethod} : \mathsf{Statements} \to \mathsf{Methods}$ yields the method of each statement.

3.2 External Components

Before running BackFlow, a static analyzer must have already inferred some information: a taint analysis, a call graph and a heap abstraction.

A taint analysis specifies which variables, at which program points, have been inferred as tainted, since they might hold unsanitized user input or sensitive data. The taint analyzer must have received a specification of the set of sources and sinks: sources are values returned by specific methods ($\mathsf{Sources} \subseteq \mathsf{Methods}$); sinks are values passed to method parameters ($\mathsf{Sinks} \subseteq \mathsf{Methods} \times \mathbb{N}$, where the second element is the index of the parameter). The taint analysis is then a predicate $\mathsf{taint} : \mathsf{VarAtStatement} \to \{\texttt{true}, \texttt{false}\}$, that holds if a given variable, after a given statement, has been inferred as tainted. We assume it sound: if there is a feasible execution of the code where variable $\texttt{x}$ actually contains a tainted value after a statement $\texttt{st}$, then $\mathsf{taint}(\texttt{st}, \texttt{x})$ must hold.

A call graph is a function $\mathsf{called} : \mathsf{Statements} \to \wp(\mathsf{Methods})$ that, given a call statement $\texttt{st}$, yields an over-approximation of the set of methods that might be called there. This is a standard component of a static analyzer [21,38]. It is handy, sometimes, to see the call graph as the inverse function $\mathsf{caller} : \mathsf{Methods} \to \wp(\mathsf{Statements})$ that, given a method $\texttt{m}$, yields an over-approximation of the set of call statements that might call it.

There are many heap abstractions [22,15]. BackFlow is agnostic about the chosen one, as long as it provides a function $\mathsf{writersVisible} : \mathsf{Statements} \to \wp(\mathsf{Statements})$ that, given a heap read $\texttt{y} = \texttt{x.f}$, yields an over-approximation of the set of heap writes $\texttt{y.f} = \texttt{x}$ where the read value might have been written, previously. The heap abstraction must also provide a must alias analysis, given as a predicate $\mathsf{alias} : \mathsf{Constraints} \to \{\texttt{true}, \texttt{false}\}$ over alias equality constraints between variables at some program points: $\mathsf{Constraints} = \mathsf{VarAtStatement} \times \mathsf{VarAtStatement}$. This predicate is extended to $\mathsf{constraintsSatisfied} : \wp(\mathsf{Constraints}) \to \{\texttt{true}, \texttt{false}\}$, by letting $\mathsf{constraintsSatisfied}(C)$ hold if and only if, for every $c \in C$, predicate $\mathsf{alias}(c)$ holds.

3.3 Flow Reconstruction

Once the supporting taint analysis has inferred a sink as potentially tainted, the backwards flow reconstructor tries to reconstruct a path of variables at statements, providing evidence about how tainted data flows, from a source, into that sink. The path is given as a *taint graph*, with vertexes in $\mathsf{VarAtStatement}$. Taint

$$(1)\qquad \mathbb{S}[\![\mathtt{y_1} = \mathtt{x_1.f}, \mathtt{y_1}]\!] = \{((\mathtt{y_2.f} = \mathtt{x_2}, \mathtt{x_2}), \{((\mathtt{y_1} = \mathtt{x_1.f}, \mathtt{x_1}), (\mathtt{y_2.f} = \mathtt{x_2}, \mathtt{y_2}))\}) : \mathtt{y_2.f} = \mathtt{x_2} \in \mathsf{writersVisible}(\mathtt{y_1} = \mathtt{x_1.f})\}$$

$$(2)\ \mathbb{S}[\![\mathtt{y_1} = \mathtt{x_1.m}(\cdots), \mathtt{y_1}]\!] = \{((\mathtt{ret\ x_2}, \mathtt{x_2}), \{((\mathtt{y_1} = \mathtt{x_1.m}(\cdots), \mathtt{x_1}), (\mathtt{ret\ x_2}, \mathtt{this}))\}) : \exists(\mathtt{pars}, \mathtt{cfg}) \in \mathsf{called}(\mathtt{y_1} = \mathtt{x_1.m}(\cdots)) : \mathtt{ret\ x_2} \in \mathtt{cfg}\}$$

$$(3)\qquad \mathbb{S}[\![\mathtt{start}, \mathtt{p_i}]\!] = \{((\mathtt{y} = \mathtt{x.m}(\mathtt{x_1}, \cdots, \mathtt{x_n}), \mathtt{x_i}), \{((\mathtt{start}, \mathtt{this}), (\mathtt{y} = \mathtt{x.m}(\mathtt{x_1}, \cdots, \mathtt{x_n}), \mathtt{x}))\} \cup \{((\mathtt{start}, \mathtt{p_j}), (\mathtt{y} = \mathtt{x.m}(\mathtt{x_1}, \cdots, \mathtt{x_n}), x_j)) : j \in [1..n]\}) : ((\mathtt{p_1}, \cdots, \mathtt{p}_n), \mathtt{cfg}) = \mathsf{getMethod}(\mathtt{start}) \wedge \mathtt{y} = \mathtt{x.m}(\mathtt{x_1}, \cdots, \mathtt{x_n}) \in \mathsf{caller}((\mathtt{p_1}, \cdots, \mathtt{p}_n), \mathtt{cfg})\}$$

$$(4)\qquad \mathbb{S}[\![\mathtt{st}, \mathtt{x}]\!] = \{((\mathtt{st}', \mathtt{x}), \emptyset) : \mathtt{st}' \in \mathsf{pred}(st)\} \text{ for any other statement.}$$

Fig. 4: A single step of backwards propagation.

graphs TG are then defined as $(\mathsf{VarAtStatement}, \mathsf{VarAtStatement} \times \mathsf{VarAtStatement})$, that is, a poset with upper bound operator $(\mathsf{v_1}, \mathsf{e_1}) \sqcup_{\mathsf{TG}} (\mathsf{v_2}, \mathsf{e_2}) = (\mathsf{v_1} \cup \mathsf{v_2}, \mathsf{e_1} \cup \mathsf{e_2})$.

In order to be context-sensitive, the reconstructor, while proceeding backwards from a sink, collects a set of alias constraints in $\wp(\mathsf{Constraints})$, that must hold if the path leading to the sink is feasible. Otherwise, the path is an artifact of the approximated taint analysis, not allowed by the heap abstraction. During this backwards procedure, the reconstructor builds the taint graph and attaches, to each of its newly discovered vertexes, the collected set of alias constraints that must hold there. Therefore, the reconstructor builds a function in $\mathsf{CV} : \mathsf{VarAtStatement} \to \wp(\mathsf{Constraints})$. A larger set of alias constraints represents a smaller set of concrete states. Hence, the upper bound operator is $\dot{\cap}$, *i.e.*, the functional lifting of set intersection: $\mathsf{c_1} \sqcup_{\mathsf{CV}} \mathsf{c_2} = \mathsf{c_1} \dot{\cap} \mathsf{c_2}$. The poset of heap constraints is then $\langle \mathsf{CV}, \sqcup_{\mathsf{CV}} \rangle$.

The reconstructor keeps a state during its execution, consisting of the currently computed taint graph and of the function mapping each of its vertexes to the alias constraints that must hold there: $\Sigma^{\#} = \mathsf{TG} \times \mathsf{CV}$. These states form a poset with upper bound operator $(\mathsf{g_1}, \mathsf{c_1}) \sqcup_{\Sigma^{\#}} (\mathsf{g_2}, \mathsf{c_2}) = (\mathsf{g_1} \sqcup_{\mathsf{TG}} \mathsf{g_2}, \mathsf{c_1} \sqcup_{\mathsf{CV}} \mathsf{c_2})$.

3.4 Backwards Propagation

This section formalizes how the flow reconstructor tracks, backwards, the value in a sink towards a source.

Figure 4 shows the rules for a single step of backwards propagation, for the language from Section 3.1, that will later be extended into a multi-steps propagation. It defines a function $\mathbb{S} : \mathsf{VarAtStatement} \to \wp(\mathsf{VarAtStatement} \times \wp(\mathsf{Constraints}))$ that, given a variable $(\mathtt{st}, \mathtt{n}) \in \mathsf{VarAtStatement}$, meaning that the reconstructor wants to follow, backwards, the value of $\mathtt{v}$ after statement $\mathtt{st}$, computes a set of variables to track after the preceding statements and a set of

alias constraints that must hold if that step is feasible. In particular, the three rules consider the following situations:

1. when tracking a field read statement, the reconstructor tracks the value assigned to the field by any possible writer, taking note of an alias constraint stating that the receiver of the field read and write must be aliased;
2. when tracking the value returned by a method call, the reconstructor tracks the value returned by any `ret` inside any method that might be called there, by using the call graph; moreover, it takes note of an alias constraint stating that the receiver of the method call and the `this` variable inside the callee must be aliased;
3. when tracking a method formal argument, once the beginning of the method has been reached, the reconstructor tracks the actual argument passed by any possible caller of the method, by using the call graph; moreover, it takes note of an alias constraint stating that `this` variable inside the method must be alias of the receiver of the method call, and that the corresponding formal and actual parameters must be aliased as well;
4. when tracking any other statement (assignments to other variables, not currently tracked, or field writes) the reconstructor simply propagates the tracked variable, backwards.

This single-step propagation does not check the feasibility of the step. Hence, the results of $\mathbb{S}$ are refined by dropping unfeasible states, through function

$$\mathsf{filter}(R) = \{(v, C) : (v, C) \in R \wedge \mathsf{taint}(v) \wedge \mathsf{constraintsSatisfied}(C)\}$$

used for the definition of the abstract state transformer:

$$\begin{aligned}
\mathbb{S}_\sigma[\![(\mathtt{st}, \mathtt{x}), ((V, E), c)]\!] = ((V', E'), c') \text{ where} \\
(1) \quad & \mathbb{S}[\![\mathtt{st}, \mathtt{x}]\!] = R \\
(2) \quad & R' = \mathsf{filter}(R) \\
(3) \quad & V' = V \bigcup_{(v', C) \in R'} \{v'\} \\
(4) \quad & E' = E \bigcup_{(v', C) \in R'} \{(v, v')\} \\
(5) \quad & c' = c[v' \mapsto (c(v') \cup C) : (v', C) \in R']
\end{aligned}$$

In this way, filter drops all the flows that are unfeasible using the results of the taint analysis. For instance, if a sanitizer is used by the program, this would produce a not-tainted result, and even if the flow reconstructor backwardly reached the result of the method call, filter would drop it.

Intuitively, given the variable `x` at a statement `st`, whose value must be tracked backwards, and an abstract state (that is, a taint graph (V, E) and a function of constraints c), it (1) applies the single-step propagation, (2) drops all unfeasible values, (3) adds the new vertexes discovered by the single-step propagation, (4) the corresponding edges connecting the predecessor and the statements produced by the single-step propagation, and (5) updates the alias constraints tracked for the given statement and tracked variable.

Flow reconstruction is defined as a least fixpoint:

$$\mathsf{BackFlow}(v_0) = lfp\lambda_{\emptyset}^{\sqsubseteq^{\#}_{\Sigma}}((V,E),c).(((\{v_0\},\emptyset),\emptyset) \sqcup_{\Sigma^{\#}} \{\mathbb{S}_{\sigma}[\![v,((V,E),c)]\!] : \mathtt{v} \in V\}$$

This fixpoint requires an initial *seed* v_0, from where the backwards reconstruction starts. This is any sink where the taint analysis issues a warning, for which the flow reconstruction is performed. The fixpoint is reached in a final number of steps, since it works over a finite domain: there is only a finite set of statements and variables, hence a finite set of vertexes and edges of taint graphs, and a finite set of alias constraints.

4 Integration with Julia

We have implemented BackFlow inside the Julia static analyzer [35], an industrial tool that analyzes Java and C# bytecode. Julia is based on abstract interpretation [11,12], performs static analysis based on denotational or constraint-based semantics, and implements a taint analyzer, various heap abstractions and a call graph builder. This section presents the latter components and how BackFlow integrates inside Julia. For the sake of readability, the formalization from Section 3 deals with source code statements. However, Julia analyzes bytecode, hence this section refers to local variables and stack values instead of program variables.

BackFlow uses Julia's taint analysis to implement function taint (Section 3.2). Julia's taint analysis models explicit information flows through Boolean formulas. Boolean variables correspond to program variables and their models are a sound overapproximation of all taint behaviors for the variables in scope at a given program point. For instance, the abstraction of bytecode `load k t`, that pushes on the operand stack the value of local variable k, is the Boolean formula $(\check{l}_k \leftrightarrow \hat{s}_{top}) \wedge U$, stating that the taintedness of the topmost stack element after this instruction (denoted by hat $\hat{}$) is equal to the taintedness of local variable k before the instruction (denoted by hat $\check{}$); all other local variables and stack elements do not change (expressed by the formula U); taintedness before and after an instruction is distinguished by using distinct hats for the variables. There are such formulas for each bytecode instruction. Instructions that might have side-effects (field updates, array writes and method calls) need some approximation of the heap, to model the possible effects of the updates. The analysis of sequential instructions is merged through a sequential composition of formulas. Loops and recursion are saturated by fixpoint. The resulting analysis is a denotational, bottom-up taint analysis, that Julia implements through efficient binary decision diagrams [6]. Julia uses a dictionary of sources (for instance, servlets input and input methods) and sinks (such as SQL query methods, command execution routines, session manipulation methods) of tainted data, so that flows from sources to sinks can be established.

Julia contains several heap abstractions. In particular, a definite aliasing analysis between variables and expressions [29], that identifies local variables and stack elements that are definitely alias. A possible sharing analysiss [34]

and a possible reachability analysis [30] between pairs of variables allow one to reason about side-effects. In addition, a creation point analysis [1] infers the program points that might have created the objects flowing to a given local variable or stack element. BackFlow relies on the latter analysis to check if two reference values might be alias, and implement the alias function (Section 3.2). In addition, BackFlow uses the definite aliasing analysis to augment the set of alias constraints. Namely, if flow reconstruction requires variables `x` and `y` to be alias at some statement `st`, and definite aliasing analysis infers that `y` and `z` are definite alias at `st`, then BackFlow adds the constraint $(\texttt{x}, \texttt{z})$.

Julia approximates the dynamic targets of method calls through standard class analysis [31], widely adopted in practice [38]. BackFlow uses this information to implement functions caller and called (Section 3.2) and to determine the predecessors of statements `start` and $\texttt{y} = \texttt{x.m}(\cdots)$.

5 Implementation

Algorithm 1 Overall algorithm of flow reconstruction

```
1: procedure TaintAnalysisWithBackFlow(program, TaintAnalysis)
2:     res ← ∅
3:     for (st, x) ∈ Julia(program, TaintAnalysis) do
4:         ((V, E), c) ← BackFlow((st, x))
5:         f ← ∅
6:         for (st′, x′) ∈ V : called(st) ∩ Sources ≠ ∅ do
7:             f ← f ∪ getPaths((V, E), (st, x), (st′, x′))
8:         res ← res ∪ {(st, x), f}
9:     return res
```

Algorithm 1 reports the flow reconstruction algorithm. Its input is a `program`; its output is a set of warnings, possibly enriched with some taint graphs. The Julia analyzer is represented as a function Julia that, given a program and a taint analysis (*e.g.*, `Injection`) returns a set of warnings represented as potentially tainted sinks, that is, pairs of a statement and a local variable, from where flows should be reconstructed. The algorithm iterates on Julia's warnings (line 3). For each warning, it applies BackFlow, inferring a taint graph. For each statement in the taint graph, that is a source (line 6), it collects some paths that connect the source to the sink (line 7, where getPaths is a function that given a graph and two vertexes in the graph returns a set of paths in the given graph that connect the two vertexes). The collected taint graphs are then associated with the warning (line 8): since the taint graph returned by BackFlow might contain no calls to sources, f might well be empty here. At the end, the algorithm returns all warnings with the corresponding taint graphs (line 9).

Algorithm 1 reflects the formalization of Section 3. However, some implementation choices have been made to support scalability. Namely, keeping alias con-

straints can be expensive, in particular because the need of computing their closure, as well as the computation of function constraintsSatisfied. Hence, BackFlow caches the results of the evaluation of alias constraints, and their closure, to avoid recomputation. Moreover, closure of alias constraints, including the constraints inferred by the definite aliasing analysis (Section 4), leads easily to an unmanageable amount of constraints. Therefore, BackFlow drops the constraints that refer to variables not in scope (*i.e*, in another method). Finally, the backwards flow reconstruction has been limited to a maximal depth n, controlled by the user. This, however, reduces the number of taintness warnings for which the reconstructor succeeds in inferring a taint graph.

BackFlow uses library JGraphT 1.3.0[6] to implement the taint graphs. The latter have been represented as `graphml` files, in hierarchical structure: nodes are grouped by package, class, method and source line of the statement. BackFlow relies on the implementation of the Dijkstra shortest-path algorithm in class `DijkstraShortestPath`, to compute the final taint graphs (function getPaths).

BackFlow has been embedded inside Julia's `Injection` checker. Figure 5 shows the UI of BackFlow, as it appears in Julia's Eclipse plugin[7]. The following options have been added to the `Injection` checker:

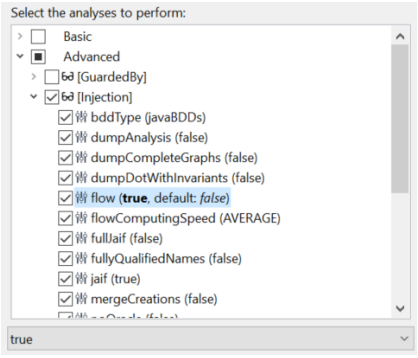


Fig. 5: BackFlow's UI in Eclipse.

1. flow (defaults to false): if true, BackFlow is used to infer the taint graphs for the warnings;
2. flowComputingSpeed (defaults to AVERAGE): specifies the maximal depth n of the inferred taint graphs: FASTEST ($n = 500$), FAST ($n = 1000$), AVERAGE ($n = 2000$), SLOW ($n = 4000$) or SLOWEST (no limit).
3. dumpCompleteGraphs (defaults to false): if true, dumps the complete backwards taint graph (that is, before the application of getPaths).

6 Experimental Results

This section presents the results of the application of BackFlow. It first analyzes the quality of these results on a specific application (how many alarms get a taint graph and which are true or false). Then it studies them quantitatively, on

[6] `https://jgrapht.org/`

[7] The user manual can be retrieved at `https://static.juliasoft.com/docs/latest/pdf/EclipsePluginUserGuide.pdf`

a well-known set of benchmarks used in previous works about taint analysis of Java programs. The experiments have been performed on a HP EliteBook 850 G4 laptop equipped with an Intel Core i7-7500 CPU at 2.7 GHz and 16 GB of RAM memory running Microsoft Windows 10 Pro and Oracle JDK version 1.8.0_141. During the analysis, 8 GB were allocated to Java.

The raw experimental results have been published in Zenodo [19]. To reproduce the results:

1. register at `https://portal.juliasoft.com`,
2. install the Julia Eclipse plugin (`https://static.juliasoft.com/docs/latest/pdf/EclipsePluginUserGuide.pdf`),
3. contact JuliaSoft (`info@juliasoft.com`) asking enough credits to run the analyses,
4. import the projects in Eclipse (in some cases it might be easier to import them as "Julia projects", check the Eclipse Plugin User Guide for details),
5. run the analyses with the configuration described in the previous section.

6.1 Qualitative Study: WebGoat

WebGoat is a "deliberately insecure web application maintained by OWASP and designed to teach web application security lessons"[8]. It is a good target to evaluate a taint analysis, since it contains a wide range of different injections (such as SQL injection or XSS) and it has been widely used as benchmark in the past.

Warning	# w. flow		# w/o flow	
	True	False	True	False
AddressInjection	1	0	0	0
CommandInjection	5	0	0	0
HttpResponseSplitting	3	0	0	3
LogForging	1	0	0	0
MessageInjection	0	0	0	1
PathInjection	6	0	0	3
ReflectionInjection	3	0	0	2
ResourceInjection	2	0	0	0
SessionInjection	2	0	0	2
SQLInjection	38	0	0	0
XPathInjection	1	0	0	0
XSSInjection	2	0	0	3
Total	64	0	0	14

Table 1: Results on WebGoat 6.0.

Julia produces 78 warnings on WebGoat 6.0. BackFlow builds a taint graph for 64 (82%) of them. Table 1 reports our manual classification of the 78 warnings, where columns "# w. flow" and "# w/o flow" report the statistics of the 64 warnings with a taint graph and of the 14 warnings without such graph, respectively. It shows that the 64 warnings for which BackFlow could reconstruct a taint graph are true alarms, while the remaining 14 are false alarms. Therefore, BackFlow's backwards flow reconstruction is an empirical evidence of a true alarm. This is an ideal result, since BackFlow was always able to discern between true and false alarms, but this is not always the case in general. In particular, BackFlow might produce a taint graph for a false alarm, for instance when dealing with code that stores tainted data into an array at some index, and then reads data from another index and passes it to a sink; and it might fail to produce a taint graph for a true alarm, because of limits of the static alias analysis engine.

[8] `https://www.owasp.org/index.php/Category:OWASP_WebGoat_Project`

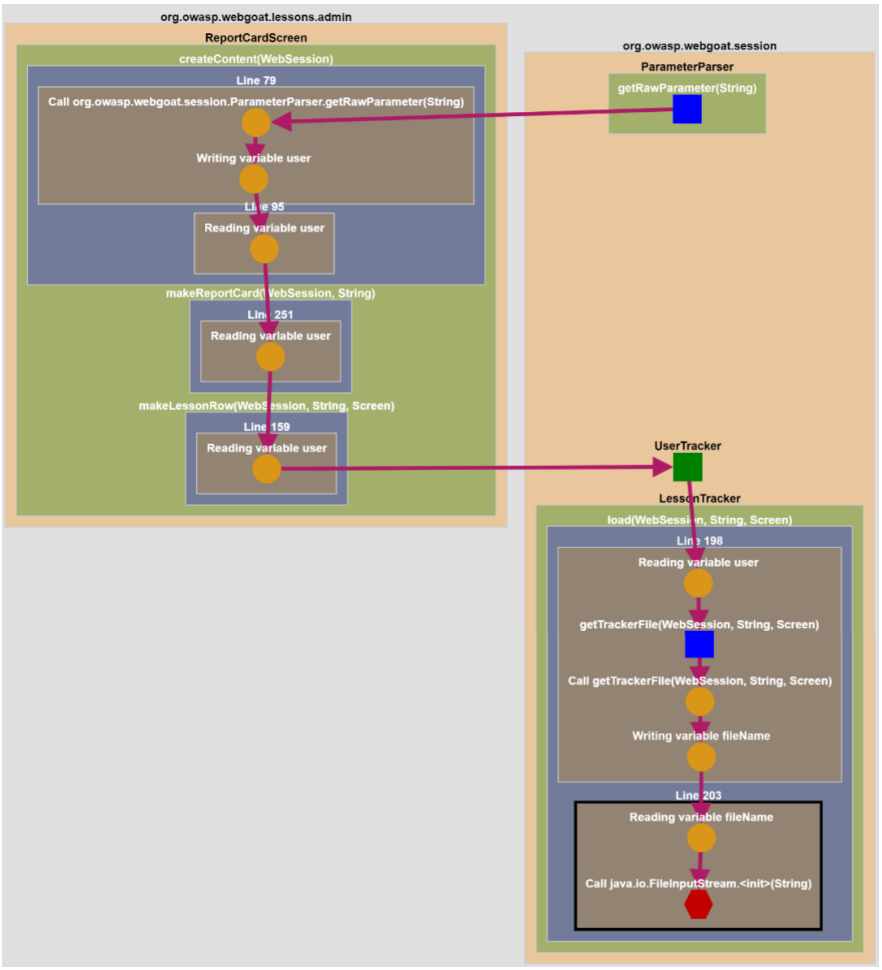


Fig. 6: The taint graph for a complex path injection warning.

6.2 True Alarms

BackFlow reconstructs taint graphs for two true alarms, allowing the user to immediately identify the source of tainted data. Manual inspection would have required relevant effort to identify these flows because of the complex structure of code.

Figure 6 shows the taint graph reconstructed for a complex path injection. Figure 7a shows the code where the flow occurs. Julia's taint analysis warns about a possible path injection at line 203 in class `LessonTracker` (corresponding to line 35 in Figure 7a). Namely, the first parameter of the `FileInputStream` constructor receives a string `fileName`, returned by method `getTrackerFile`, that consumes the three parameters of method `load`. The taint analysis detects

```
1  class ReportCardScreen {
2    Element createContent(WebSession s) {
3      String user = s.getParser()
4          .getRawParameter(USERNAME);
5      ec.addElement(makeReportCard(s, user));
6    }
7    Element makeReportCard
8        (WebSession s, String u) {
9      t.addElement(makeLessonRow(s, u, screen));
10   }
11   private TR makeLessonRow
12       (WebSession s, String u, Screen screen) {
13     LessonTracker lessonTracker = UserTracker.
14       instance().getLessonTracker(s, u, screen);
15   }
16 }
17 class UserTracker{
18   public LessonTracker getLessonTracker
19       (WebSession s, String u, Screen screen) {
20     LessonTracker tracker =
21       LessonTracker.load(s, u, screen);
22   }
23 }
24 class LessonTracker{
25   static String getTrackerFile
26     (WebSession s, String user, Screen screen) {
27     return getUserDir(s) + user + "." +
28       screen.getClass().getName() + ".props";
29   }
30   LessonTracker load
31     (WebSession s, String user, Screen screen) {
32     String fileName =
33       getTrackerFile(s, user, screen);
34     FileInputStream in =
35       new FileInputStream(fileName);
36   }
37 }
```

(a) Snippet of the code for Figure 6.

```
1  class LessonAdapter extends AbstractLesson { ... }
2  class ThreadSafetyProblem extends LessonAdapter {
3    protected Element createContent(WebSession s) {
4      ElementContainer ec = new ElementContainer();
5      currentUser = s.getParser()
6        .getRawParameter(USER_NAME, "");
7      originalUser = currentUser;
8      ec.addElement("Account information for user: "+
9                originalUser + "<br><br>");
10     return ec;
11   }
12 class AbstractLesson extends Screen {
13   public void handleRequest(WebSession s) {
14     Form form = new Form(getFormAction(),
15       Form.POST).setName("form").setEncType("");
16     form.addElement(createContent(s));
17     setContent(form);
18   }
19 }
20 class Screen {
21   private Element content;
22   protected void setContent(Element content) {
23     this.content = content;
24   }
25   public String getContent() {
26     return content.toString();
27   }
28   public void output(PrintWriter out) {
29     out.print(getContent());
30   }
31 }
```

(b) Snippet of the code for Figure 8.

Fig. 7: Snippets of codes for two true alarms.

`user` as tainted, and then BackFlow tracks it backwards through `UserTracker.getLessonTracker` (line 21 of Figure 7a), `ReportCardScreen.makeLessonRow` (line 14), `ReportCardScreen.makeReportCard` (line 9) and `ReportCardScreen.createContent` (line 5). There, `user` is assigned a raw servlet parameter, hence a source has been reached (line 3, where the parameter is returned by a method of WebGoat that calls the Java servlet API, but we omit this detail here). This example shows that BackFlow infers a taint graph that helps the programmer understand and hence fix the injection. Manual tracking of tainted data would be much harder. For instance, `UserTracker.getLessonTracker` is called five times in WebGoat: the programmer would have needed to check all of them to discover the one that taints `user`.

Figure 8 shows another taint graph produced by BackFlow, whose corresponding code is in Figure 7b. Julia reports an XSS warning at line 201 of class `Screen` (line 29 in Figure 7b), where field `content` is passed to `PrintWriter.print`, through a getter method. The setter method `setContent` for this field is called

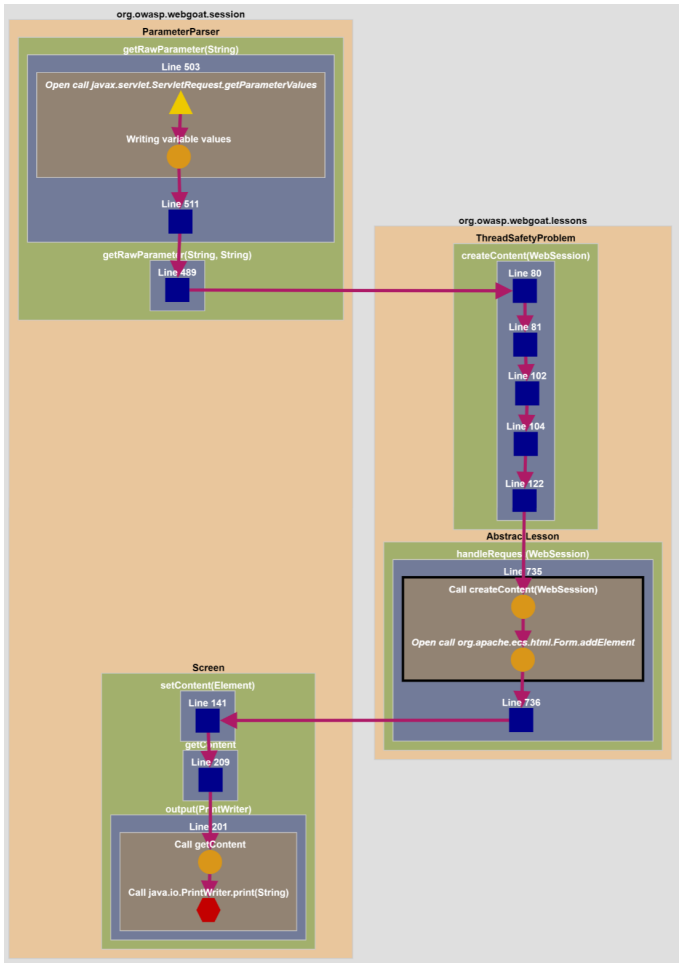


Fig. 8: The taint graph for an XSS warning.

at 19 different places in WebGoat: manual inspection would be extremely difficult. Instead, the taint graph in Figure 8, which is one of the eight different taint graphs, each for a different source, reconstructed for this warning, shows that the call from class `AbstractLesson` (that extends `Screen` and thus can call the protected `setContent`) at line 736 (line 17 in Figure 7b) passes tainted data coming from `createContent`. The latter is an abstract method in class `Screen`, implemented in 79 classes of WebGoat. The taint graph shows that an implementation passing tainted data is in class `ThreadSafetyProblem` (that extends `LessonAdapter`, that in turn extends `AbstractLesson`). This accesses a servlet parameter at line 80 (line 6 of Figure 7b) and, after some computation, adds it at line 122 to the element returned by the method.

```
1  class Encoding {
2    static String base64Encode(String str) {
3      byte[] b = str.getBytes();
4      return encoder.encode(b);
5    }
6  }
7  class Challenge2Screen {
8    private String user = "youaretheweakestlink";
9    protected Element doStage1(WebSession s) {
10     Cookie newCookie = new Cookie
11         (USER, Encoding.base64Encode(user));
12         s.getResponse().addCookie(newCookie);
13   }
14 }
```

(a) False HTTP response splitting.

```
1  class Course {
2    List<String> files = new LinkedList<>();
3    void loadFiles(ServletContext context, String path) {
4      Set resourcePaths = context.getResourcePaths(path);
5      Iterator itr = resourcePaths.iterator();
6      while (itr.hasNext())
7        files.add(itr.next());
8      }
9      private void loadResources() {
10     for (String absoluteFile : files) {
11       String fileName = getFileName(absoluteFile);
12       ...
13     }
14   }
15   private static String getFileName(String s) {
16     String fileName = new File(s).getName();
17     //Some computations on fileName
18     return fileName;
19   }
20 }
```

(b) False path injection.

Fig. 9: Snippets of code for the false alarms.

6.3 False Alarms

BackFlow fails to reconstruct a flow graph for two injection warnings reported by Julia's taint analysis. They turn out to be false alarms.

Julia warns about a potential HTTP response splitting at line 172 of class `Challenge2Screen`. Figure 9a reports the corresponding source code (see line 12). Julia's taint analysis infers that `Encoding.base64Encode` returns a tainted value, which taints variable `newCookie`. However, field `user` is not tainted and the code of `Encoding.base64Encode` is such that, if its parameter is not tainted, then also the returned value is not tainted. But Julia's taint analysis infers that the returned value could be tainted because `encoder` has class `sun.misc.BASE64Encoder`, not available to the analysis. Hence a worst-case assumption on missing code [36] is applied. BackFlow reconstructs the flow until the read of the static field `encoder` and stops there, since it does not find the origin of the tainted data.

Julia reports a possible path injection warning at line 83 of class `Course`. Figure 9b reports the corresponding source code (see line 16). Julia sees that method `Course.loadResources` might call `getFileName` with a tainted parameter (line 11), during the iteration over all file paths in the list `files`. This field is initialized to an empty list of strings (line 2) and only method `loadFiles` adds elements to it (line 7) from the parameters obtained from the resource paths of the servlet context. Julia taint analysis infers that elements of `files` might be tainted, but the only statement that adds elements to the list (line 7) does not deal with tainted data. BackFlow reconstructs the flow only until the beginning of the for each loop at line 10. The false alarm is due to the fact that Julia's taint analysis sees the receiver of the `getfield files` bytecode statement as tainted.

Program			Time (sec)		Other		Http		Log		Path		Refl		Sess		SQL		URL		XSS		**Total**	
Name	**Ver.**	**LOC**	Taint	Rec	w/o	w	w/o	w	w/o	w	w/o	w	w/o	w	w/o	w	w/o	w	w/o	w	w/o	w	w/o	w
blojsom	3.3b	17144	268	19	0	1	2	6			17	19			1	17			0	6	1	0	21	49
bluebog	0.9	1930	137	3							6	4											6	4
friki	1.3.0	7718	127	107							0	4	1	3	1	2							2	9
gestcv	1.0.0	3948	93	3							0	1									1	0	1	1
jboard	0.30	3397	143	3											0	1	11	1					11	2
jspwiki	2.11	30024	412	192	1	0	3	8	96	114	19	22	5	8	1	1			0	1	8	22	133	176
jugjobs		869	94	2	1	0					2	0	1	0			4	0					8	0
pebble	2.6.4	23124	581	55			1	6			69	29	13	0	0	3			1	1	3	12	87	51
personal	vel.	3480	148	4			0	1			2	0	5	0			1	0	2	0			10	1
photov	2.1	9368	115	4					3	2					1	39	3	0					7	41
roller	0.9.6	11202	42	7			0	1	10	1	5	0			6	2					8	3	29	7
snipsnap	0.7	3736	96	2																			0	0
Total		115940	2256	401	2	1	6	22	109	117	120	79	25	11	10	65	19	1	3	8	21	37	315	341

Table 2: Experimental results on standard benchmarks.

6.4 Quantitative Study: Benchmarks

BackFlow has been run on a set of Java web applications used as benchmarks to evaluate similar tools, in previous work. In particular, this set is taken from [23], that collected these applications starting from other previous work [26,37,41]. The goal of this quantitative experiment is to study the scalability of BackFlow and see if it is as precise as on WebGoat.

Table 2 reports the results. For each application, it reports its version (column Vers.)[9], its number of lines of code (LOC, as estimated by Julia), the time for taint analysis and for BackFlow (in seconds), and the number of warnings with (column w) or without (column w/o) taint graph. This figure reports numbers only for the types of warnings that were actually raised by Julia: Http stands for Http response splitting, Log for log forging, Path for path injections, Refl for reflection injections, Sess for session injections, SQL for SQL injections, URL for URL injections, XSS for cross site scripting and Other for all remaining types of injection warnings - address injections for blojsom, DOS injections for jspkiwi and message injections for jugjobs.

All together, the analyzed applications consist in about 116KLOCs. Julia analysis and BackFlow took 37'26" and 6'41", respectively. Hence, BackFlow requires less than a fifth of the overall analysis time and it scales to real world applications.

Out of 656 injection warnings, BackFlow builds a taint graph for 341 (52%). While there are significant differences between different types of applications and warnings (for instance, BackFlow reconstructs 87% of session injection flows, but only 5% of SQL injection flows), the overall result shows that BackFlow is effective in building taint graphs for injection warnings in real world applications. The efficacy is smaller than on the qualitative study (where BackFlow reconstructs a taint graph for 82% of the warnings). This difference is possibly justified by the fact that WebGoat's didactic code is more regular than that of these benchmark applications.

[9] We were not able to find a distribution of `jugjobs` with a version number.

7 Conclusion

BackFlow proves to be able to reconstruct the flow of tainted data, as taint graphs. Experimental results show that the fact that BackFlow provides (or not) a taint graph for a warning is a clear empirical indication that the alarm is true (respectively, false). Moreover BackFlow scales to real-world applications.

References

1. Andersen, L.: Program analysis and specialization for the C programming language. Phd thesis, University of Copenhagen (1994)
2. Arkin, B., Stender, S., McGraw, G.: Software penetration testing. IEEE Security Privacy **3**(1), 84–87 (2005)
3. Arzt, S., Rasthofer, S., Fritz, C., Bodden, E., Bartel, A., Klein, J., Le Traon, Y., Octeau, D., McDaniel, P.: FlowDroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In: Proceedings of PLDI '14. ACM (2014)
4. Balzarotti, D., Cova, M., Felmetsger, V., Jovanovic, N., Kirda, E., Kruegel, C., Vigna, G.: Saner: Composing static and dynamic analysis to validate sanitization in web applications. In: Proceedings of S&P '08. IEEE (2008)
5. Barbon, G., Cortesi, A., Ferrara, P., Pistoia, M., Tripp, O.: Privacy analysis of android apps: Implicit flows and quantitative analysis. In: Proceedings of CISIM '15. LNCS, Springer (2015)
6. Bryant, R.: Symbolic Boolean Manipulation with Ordered Binary-Decision Diagrams. ACM Computing Survey **24**(3), 293–318 (1992)
7. Buro, S., Mastroeni, I.: Abstract code injection. In: Proceedings of VMCAI '18. Springer (2018)
8. Clause, J., Li, W., Orso, A.: Dytan: A generic dynamic taint analysis framework. In: Proceedings of ISSTA '07. ACM (2007)
9. Cortesi, A., Ferrara, P., Pistoia, M., Tripp, O.: Datacentric semantics for verification of privacy policy compliance by mobile applications. In: Proceedings of VMCAI '15. LNCS, Springer (2015)
10. Costantini, G., Ferrara, P., Cortesi, A.: A suite of abstract domains for static analysis of string values. Software: Practice and Experience **45**(1) (Feb 2015)
11. Cousot, P., Cousot, R.: Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In: Proceedings of POPL '77. ACM (1977)
12. Cousot, P., Cousot, R.: Systematic Design of Program Analysis Frameworks. In: Proceedings of POPL '79. ACM (1979)
13. Csallner, C., Smaragdakis, Y.: Check 'n' Crash: Combining static checking and testing. In: Proceedings of ICSE '05. ACM (2005)
14. Denning, D.E.: A lattice model of secure information flow. Commun. ACM **19**(5), 236–243 (May 1976)
15. Deutsch, A.: Interprocedural may-alias analysis for pointers: Beyond k-limiting. In: Proceedings of PLDI '94. ACM (1994)
16. Enck, W., Gilbert, P., Han, S., Tendulkar, V., Chun, B.G., Cox, L.P., Jung, J., McDaniel, P., Sheth, A.N.: TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones. ACM Trans. Comput. Syst. **32**(2), 5:1–5:29 (Jun 2014)

17. Ferrara, P.: Generic combination of heap and value analyses in abstract interpretation. In: Proceedings of VMCAI '14. LNCS, Springer (2014)
18. Ferrara, P.: A generic framework for heap and value analyses of object-oriented programming languages. Theoretical Computer Science **631** (2016)
19. Ferrara, P., Olivieri, L., Spoto, F.: BackFlow: Backward Context-sensitive Flow Reconstruction of Taint Analysis Results (Nov 2019). https://doi.org/10.5281/zenodo.3539240, `https://doi.org/10.5281/zenodo.3539240`
20. Ferrara, P., Tripp, O., Pistoia, M.: MorphDroid: Fine-grained privacy verification. In: Proceedings of ACSAC '15. ACM (2015)
21. Grove, D., DeFouw, G., Dean, J., Chambers, C.: Call graph construction in object-oriented languages. In: Proceedings of OOPSLA '97. ACM (1997)
22. Hind, M.: Pointer analysis: Haven't we solved this problem yet? In: Proceedings of PASTE '01. ACM (2001)
23. Huang, W., Dong, Y., Milanova, A.: Type-based taint analysis for java web applications. In: Proceedings of FASE '14. Springer (2014)
24. Jovanovic, N., Kruegel, C., Kirda, E.: Pixy: a static analysis tool for detecting web application vulnerabilities. In: Proceeding of S&P '06. IEEE (2006)
25. Leek, T.R., Brown, R.E., Zhivich, M.A., Leek, T.R., Brown, R.E.: Coverage maximization using dynamic taint tracing. Tech. rep., MIT Lincoln Laboratory (2007)
26. Livshits, V.B., Lam, M.S.: Finding security vulnerabilities in java applications with static analysis. In: Proceedings of USENIX Security '05. USENIX Association (2005)
27. Myers, A.C.: JFlow: Practical mostly-static information flow control. In: Proceedings of POPL '99. ACM (1999)
28. Newsome, J., Song, D.: Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In: Proceedings of NDSS '05. The Internet Society (2005)
29. Nikolic, D., Spoto, F.: Definite expression aliasing analysis for Java bytecode. In: Proceedings of ICTAC '12. Springer (2012)
30. Nikolic, D., Spoto, F.: Reachability analysis of program variables. ACM Trans. Program. Lang. Syst. **35**(4), 14:1–14:68 (Jan 2014)
31. Palsberg, J., Schwartzbach, M.I.: Object-oriented type inference. In: Proceedings of OOPSLA '91. ACM (1991)
32. Reps, T., Horwitz, S., Sagiv, M.: Precise interprocedural dataflow analysis via graph reachability. In: Proceedings of POPL '95. ACM (1995)
33. Sabelfeld, A., Myers, A.C.: Language-based information-flow security. IEEE J.Sel. A. Commun. **21**(1), 5–19 (Sep 2006)
34. Secci, S., Spoto, F.: Pair-sharing analysis of object-oriented programs. In: Proceedings of SAS '05. Springer (2005)
35. Spoto, F.: The Julia Static Analyzer for Java. In: Proceedings of SAS '16. LNCS, Springer (2016)
36. Spoto, F., Burato, E., Ernst, M.D., Ferrara, P., Lovato, A., Macedonio, D., Spiridon, C.: Static identification of injection attacks in java. ACM Transactions on Programming Languages and Systems (TOPLAS) **41** (July 2019)
37. Sridharan, M., Artzi, S., Pistoia, M., Guarnieri, S., Tripp, O., Berg, R.: F4F: Taint analysis of framework-based web applications. In: Proceedings of OOPSLA '11. ACM (2011)
38. Tip, F., Palsberg, J.: Scalable propagation-based call graph construction algorithms. In: Proceedings of OOPSLA '00. ACM (2000)

39. Tripp, O., Ferrara, P., Pistoia, M.: Hybrid security analysis of web JavaScript code via dynamic partial evaluation. In: Proceedings of ISSTA '14. ACM (2014)
40. Tripp, O., Guarnieri, S., Pistoia, M., Aravkin, A.: ALETHEIA: Improving the usability of static security analysis. In: Proceedings of CCS '14. ACM (2014)
41. Tripp, O., Pistoia, M., Fink, S.J., Sridharan, M., Weisman, O.: TAJ: Effective taint analysis of web applications. In: Proceedings of PLDI '09. ACM (2009)
42. Vogt, P., Nentwich, F., Jovanovic, N., Kirda, E., Kruegel, C., Vigna, G.: Cross-site scripting prevention with dynamic data tainting and static analysis. In: Proceedings of NDSS '05. The Internet Society (2007)
43. Wassermann, G., Su, Z.: Sound and precise analysis of web applications for injection vulnerabilities. In: Proceedings of PLDI '07. ACM (2007)
44. Yin, H., Song, D., Egele, M., Kruegel, C., Kirda, E.: Panorama: Capturing system-wide information flow for malware detection and analysis. In: Proceedings of CCS '07. ACM (2007)
45. Zanioli, M., Ferrara, P., Cortesi, A.: SAILS: static analysis of information leakage with sample. In: Proceedings of SAC '12. ACM (2012)