Taint Trace Analysis For Java Web Applications

Yaju Li, Chenyi Zhang
Jinan University, Guangzhou, China

Qin Li
East China Normal University, Shanghai, China

***Abstract*—Taint analysis is concerned about whether a value in a program can be influenced, or tainted, by user input. Existing works on taint analysis focus on tracking the propagation of taint flows between variables in a program, and a security risk is reported whenever a *taint source* (user input) flows to a *taint sink* (resource that requires protection). However, a reported bug may have its taint source and taint sink located in different software components, which complicates the bug tracking and bug confirmation for developers. In this paper, we propose *Taint Trace Analysis* (TTA), which extends P/Taint, a context-sensitive Java taint analysis project, by making the taint information flow explicit. Thanks to the underlying Datalog semantics, we describe a way to extract traces of taint flows across program contexts and field accesses in the Doop framework. Different from existing works that produce only source-sink pairs, the output of TTA can be visualized as a set of traces which illustrate the inter-procedural taint propagation from taint sources to their corresponding sinks. As a consequence, TTA provides more useful information for developers and users after a vulnerability is reported. Our implementation is also efficient, and as shown in our experiment, it adds only a small run-time overhead on top of P/Taint for a range of analyses with different types of context-sensitivities applied.**

***Index Terms*—Taint analysis, Automatic trace generation, Program analysis, Java web application**

I. Introduction

Security vulnerabilities, which are software bugs exploitable by attackers, have been long standing challenges in software security and cyber security. As an example, Worm programs, such as the Microsoft SQL server Slammer [15], the Sun Telnet worm [26] and the Stuxnet worm [3], exploit software vulnerabilities in client or server programs and gain access to hundreds of thousands of new systems (including mobile phones [24]) within minutes. Of those notorious top security risks published by OWASP [17], Injection and Cross-Site Scripting (XSS) are usually triggered by crafted user input strings that are propagated through web application to reach their victims without censorship.

Taint analysis is an indispensable weapon in our combat against security vulnerabilities in system software, network software, and mobile applications. Existing approaches, including static taint analysis [18], [19] and dynamic taint analysis [16], [6], are implemented in tools such as TAJ [25], F4F [22] and Parfait [5] for tracking taint flows in web applications written in Java and JavaScript. However, provided that a complicated flow of taint may potentially pass through a number of program contexts via method calls and returns, the existing tools report only the taint sources with their corresponding taint sinks, which do not fully reveal all useful details that would guide a developer to identify or locate security vulnerabilities with ease.

Datalog has been used in program analysis since late 1990s [9]. In Datalog, program information is initially extracted into a group of base facts, from which advanced properties can be specified and subsequently computed in the style of logic programming. Such a reasoning pattern allows us to explicitly encode a trace of taint flow from the outputs of the existing points-to analysis and taint analysis facilities in the Doop framework [2]. Therefore, in this paper, we re-encode and extend the taint analysis constraints of P/Taint in order to support explicitly exporting a trace of information flow, taking advantage of Doop and its underlying Soufflé Datalog engine [11], and provide more useful information for developers when it is required to trace a reported vulnerability. In this paper, we have made the following contributions.

- We propose a method that explicitly exports a collection of taint traces associated with a given taint source-sink pair produced from a Java web application. The reported traces are context-sensitive, i.e., they contain all inter-procedural information of the complete taint flows.
- We have implemented our algorithm and conducted experiment on Securibench Micro [13]. The experiment results have confirmed that we only added a small overhead on top of the existing P/Taint [8] implementation in Doop.

The rest of the paper is organized as follows. In Section II, we illustrate how the proposed method works in a code snippet. Section III formally introduces the Taint Flow Analysis, with the implementation and experiment works presented in Section IV. The related works are discussed in Section V, and Section VI concludes our work.

II. A Motivating Example

The following example is simplified and adapted from Securibench Micro [13], as shown in Fig. 1. Note that in this example, a variable that points to the HttpServeletRequest object is set to receive data from client (user) input, and a reference to the HttpServeletResponse is used to write back to client. In particular, at line 3 of doGet() method, a user provided value is read from client and stored in variable name, which is then passed as a parameter in a call to method foo() at

*Corresponding email: chenyi_zhang@jnu.edu.cn.
This work is partially supported by the National Natural Science Foundation of China under Grant 62077028, Guangdong Natural Science Foundation under Grant 2019KTSCX010, Guangdong Basic and Applied Basic Research Foundation under Grant 2021A1515011873, Science and Technology Planning Project of Guangzhou under Grant 202102080307, and the Project of Guangxi Key Laboratory of Trusted Software (No. kx202007).
DOI reference number: 10.18293/SEKE2022-161

```
1  public class Example {
2    protected void doGet(HttpServletRequest req,
       HttpServletResponse resp) throws IOException {
3      String name = req.getParameter(''name'');
4      foo(name, ''abc'', req, resp);
5    }
6    ...
7    void foo(String str1, String str2,
       HttpServletRequest req, HttpServletResponse
       resp) throws IOException {
8      Data d = new Data();
9      d.value1 = str1;
10     d.value2 = str2;
11     PrintWriter writer = resp.getWriter();
12     writer.println(d.value1);    /* BAD */
13     writer.println(d.value2);    /* OK */
14   }
15 }
16 ...
17 class Data {
18   String value1;
19   String value2;
20 }
```

Fig. 1. A motivating example

line 4. P/Taint will report that there is a potential *Leak* at line 12, where a sink method println() is invoked with a tainted parameter d.value1. Since the tainted value is directly sent back to user, this could potentially form an XSS vulnerability, as the request information may consist of executable malicious code contained in a crafted link that a victim is fooled to click.

In order to trace a problem reported by a taint analysis and rule out false positives, a developer usually needs to manually inspect the source code and track the associated taint flows throughout the program. Such a procedure can be tedious and error-prone, as modern day web applications often consist of numerous software packages with included code written by third-party developers. Moreover, for object-oriented programming languages such as Java, a taint flow typically consists of a serial of inter-procedural method calls as well as field access of a value via alias references under different contexts, which further complicates the job of a code inspector. Taking a closer look at the reported *Leak* at line 12 in Fig. 1, the value loaded from d.value1 is previously stored and passed as a parameter at line 4 in a call to method foo, such that we have a tainted value passing through inter-procedural value-flow, stored in heap before being loaded back and forwarded to a sink method. For larger programs, a taint trace could be more complicated. In this case, Taint Trace Analysis (TTA) is able to automatically produce a corresponding taint trace (shown in Fig. 2) and visualize the flow of values, for a better comprehension and confirmation of reported security vulnerabilities.

The produced trace in Fig. 2 has six components, each written in the form of ⟨class : method(signature)/reference⟩. The first component is HttpServletRequest.getParameter(), a pre-defined *taint source*, with its value passed to the second component, a local variable name (line 3 in the code) of type string. The second component then passes the flow to

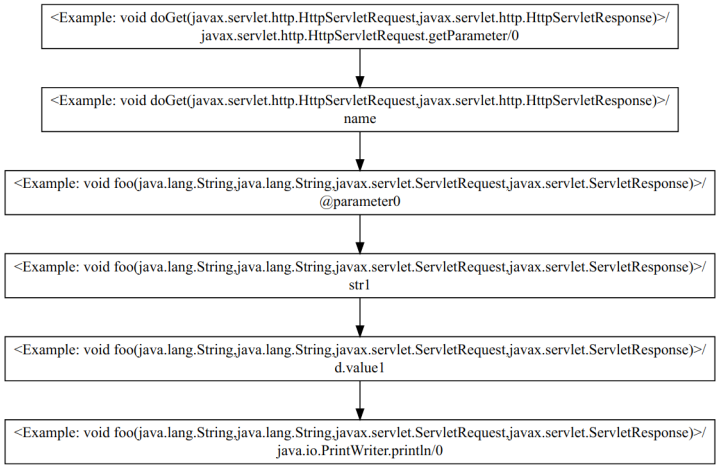

Fig. 2. A taint trace for the vulnerability reported for source code in Fig. 1.

the third component which is the first parameter of method foo (line 4 in the code), represented by @parameter0 in the trace. Eventually the flow goes to the last component println() which is a pre-defined *taint sink* (line 12 in the code).

III. Context Sensitive Taint Trace Analysis

We describe our algorithm on a simplified version of Java, with its syntax shown in Fig. 3.[1] A unique entry method such as main() is always assumed to appear in one of the classes in a well formed program. For each method, we assume that all local variables are defined at the beginning of the method, and there is always a dedicated local variable to be returned at the end. For an object allocation statement $x = \mathsf{new}^o\ A$, we assume that o uniquely identifies the allocation site. Given a program from the language, we define VAR for the set of variables, OBJ for the set of (abstract heap) objects, $\mathcal{C}$ for the set of classes, FLD for the set of fields, METHOD for the set of method, SIG for the set of method signatures, and INST the set of instructions for uniquely identifying call-sites.

We build our work on top of P/Taint [8] which is part of the program analysis framework Doop [2]. In order to generate facts for Datalog based reasoning, Doop makes use of the front end of Soot, a compiler framework for Java, for extraction of the program structural relations and basic flow relations, as shown in Table I. The extracted patterns are assumed to be self-explained on the right hand side of each relation of the table. For example, $\textsc{Alloc}(x, o)$ means there exists a statement $x = \mathsf{new}^o\ A$, which allocates object o to local variable x. Note that at the same time, $\textsc{HeapType}(o, A)$ is also added to the relation indicating that the class type of object o is A. Similarly, the existence of a relation $\textsc{Assn}(m, x, y)$ indicates that there is a statement $x = y$ in method m.

A. Taint Analysis with Points-to Analysis

First, we briefly review the P/Taint [8] project on which our work is based. At its foundation, Andersen-styled subset-based

[1]The actual implementation that handles the full Java language is more involved, and most of the implementation details are elided in this paper.

TABLE I
THE BASIC FLOW RELATIONS GENERATED FROM A PROGRAM.

ALLOC (v : VAR, o : OBJ)	// $v = \mathsf{new}^o\ A$
ASSN (m : METHOD, to : VAR, $from$: VAR)	// $to = from$ in method m
LOAD (to : VAR, $base$: VAR, f : FLD)	// $to = base.f$
STORE ($base$: VAR, f : FLD, $from$: VAR)	// $base.f = from$
VCALL ($base$: VAR, m : SIG, $inst$: INST)	// ... $= base.m(...)$ in instruction $inst$
FORMALARG ($meth$: METHOD, $i : \mathbb{N}$, v : VAR)	// v is the i-th formal arg of $meth$
ACTUALARG ($inst$: INST, $i : \mathbb{N}$, v : VAR)	// v is the i-th actual arg at callsite $inst$
FORMALRET ($meth$: METHOD, v : VAR)	// return v is a statement in $meth$
ACTUALRET ($inst$: INST, v : VAR)	// $v = base.m(...)$ in instruction $inst$
THISVAR ($meth$: METHOD, $this$: VAR)	// "$this$" as a special variable in $meth$
HEAPTYPE (o : OBJ, $A : \mathcal{C}$)	// object o is class A
LOOKUP ($A : \mathcal{C}$, m : SIG, $meth$: METHOD)	// through signature m, one can find the // corresponding method $meth$ in class A

$$
\begin{array}{lcl}
C & ::= & \mathsf{class}\ A()\ [\mathsf{extends}\ B]\ \{\overline{F};\ \overline{M}\} \\
F & ::= & A\ f \\
D & ::= & A\ z \\
M & ::= & m(\overline{x})\ \{\overline{D}; s; \mathsf{return}\ y\} \\
s & ::= & x{=}\mathsf{new}^o\ A \mid x{=}e \mid x.f{=}y \mid s; s \\
e & ::= & \mathsf{null} \mid x \mid x.f \mid x.m(\overline{y}) \\
prog & ::= & \overline{C}
\end{array}
$$

Fig. 3. Abstract syntax for the core language.

TABLE II
THE BASIC POINTS-TO RELATIONS

VARPT (v : VAR, o : OBJ)	// v points to object o
FLDPT (o_1 : OBJ, f : FLD, o_2 : OBJ)	// $o_1.f$ points to o_2

VARPT (v, o)	ALLOC (v, o).
VARPT (v_1, o)	ASSN (_, v_1, v_2), VARPT (v_2, o).
VARPT (v, o_1)	LOAD (v, $base$, f), VARPT ($base$, o_2), FLDPT (o_2, f, o_1).
FLDPT (o_1, f, o_2)	STORE (v_1, f, v_2), VARPT (v_1, o_1), VARPT (v_2, o_2).

points-to analysis [1] is applied for generating the points-to relations. For object-oriented languages such as Java, two relations are generated, VARPT, which assigns a set of objects to a variable (or reference), and FLDPT, which represents the relationship between (abstract) heap objects via fields. As shown in Table II, each relation given in the left hand side column is defined by conditions specified in the same row of the right hand side column. Taking VARPT, initially, v points to o if object o is allocated to v. The second rule extends the relation by adding $\text{VARPT}(v_1, o)$ if there is an assignment $v_1 = v_2$ and v_2 points to o (i.e., $\text{VARPT}(v_2, o)$ is already in the relation). Likewise, the last rule adds $\text{FLDPT}(o_1, f, o_2)$ if o_1 is pointed by variable v_1, o_2 is pointed by v_2, and $v_1.f = v_2$ is a (store) statement in the program.

Taint analysis is integrated with points-to analysis in P/-Taint [8], based on the fact that taint flow can be treated in the way similar to value flow through which an object is passed to a variable. We walk through the following reasoning patterns for taint analysis as shown in Table III. The first rule defines that if method m is a taint source and instruction i calls m with a value returned to variable to, then there is a taint flow to variable to, where TAINT is a constructor that introduces taint values in a new domain TAINTVALUE.[2] This definition plays the same role as the first rule (with $\text{ALLOC}(v, o)$) in Table II, since both rules are the base cases for their corresponding recursive relations, VARPT and TFLOW, respectively. The second rule in Table III resembles the second rule in Table II, by extending relations with assignments. Finally, the last rule in Table III defines the LEAK relation reporting that a taint value has reached a sink method, where $\text{SINK}(m, n)$ states that the n-th parameter of method m is a pre-defined *sink* location, and i is a call-site instruction that calls m with its n-th parameter v reachable from a taint source.

TABLE III
A FEW SAMPLE TAINT FLOW RULES

TFLOW (x : VAR, v : TAINTVALUE)	// taint value v flows to x
LEAK (v : TAINTVALUE, $inst$: INST)	// taint value v flows to // sink location $inst$

TFLOW (to, $value$)	SOURCE (m, $type$), CALLGRAPH (i, m), ACTUALRET (i, to), TAINT(i, $type$) = $value$.
TFLOW (v_1, $value$)	ASSN (_, v_1, v_2), TFLOW (v_2, $value$).
LEAK ($value$, i)	CALLGRAPH (i, m), SINK (m, n), ACTUALARG (i, n, v), TFLOW (v, $value$).

Doop also supports a range of context-sensitivity options to strengthen its points-to and taint flow computation, such as call-site-sensitivity [20], object-sensitivity [14], [12], and type-sensitivity [21]. The enhanced VARPT relation has four components.

$$\text{VARPT}(c_1 : \text{CTX},\ v : \text{VAR},\ c_2 : \text{HCTX},\ o : \text{OBJ})$$

where c_1 is the context for variable v and c_2 is the context for object o. The variable context domain CTX and the heap context domain HCTX may be defined as distinct sets. Similarly, we have the following definition for the field points-to relation.

$$\text{FLDPT}\ (c_1 : \text{HCTX},\ o_1 : \text{OBJ},\ f : \text{FLD},\ c_2 : \text{HCTX},\ o_2 : \text{OBJ})$$

[2]Here the $\text{CALLGRAPH}(i, m)$ relation is taken as the context insensitive version of the CALLGRAPH relation in Table IV. Intuitively, we retrieve the class type of the object pointed-to by the receiver variable at the call-site i, from which we match the signature of method m.

TABLE IV
SAMPLE RULES FOR k-OBJECT-SENSITIVE CONTEXT EXTENSION

REACHABLE (c : CTX, m : METHOD)
// method m is reachable with context c
CALLGRAPH (c_1 : CTX, i : INST, c_2, m : METHOD)
// method m is called at call-site i

REACHABLE ([ ], main).	
REACHABLE (c_2, m), CALLGRAPH (c_1, i, c_2, m)	REACHABLE (c_1, m_0), CONTAIN(m_0, i), VCALL (x, m', i), NEWCTX (c_1, o) = c_2, VARPT (c_1, x, c_3, o), HEAPTYPE (o, A), LOOKUP (A, m', m).

TABLE V
RELATIONS FOR ONE-STEP VALUE FLOW

// variable $from$ in context c_2 flows to variable to in context c_1
VFLOW (c_1: CTX, to : VAR, c_2: CTX, $from$: VAR)
// variable v_1 in context c_1 and v_2 in context c_2 are in the "may alias" relation
ALIAS (c_1: CTX, v_1 : VAR, c_2: CTX, v_2 : VAR)

ALIAS (c_1, v_1, c_2, v_2)	VARPT (c_1, v_1, c_3, o), VARPT (c_2, v_2, c_3, o).
VFLOW (c, to, c, $from$)	REACHABLE (c, m), ASSN (m, to, $from$).
VFLOW (c_1, x_1, c_2, x_2)	REACHABLE (c_1, m_1), LOAD (m_1, x_1, v_1, f), REACHABLE (c_2, m_2), STORE (m_2, v_2, f, x_2), ALIAS (c_1, v_1, c_2, v_2).
VFLOW (c_1, x_1, c_2, x_2)	CALLGRAPH (c_2, i, c_1, m), FORMALARG (m, n, x_1), ACTUALARG (i, n, x_2).
VFLOW (c_1, x_1, c_2, x_2)	CALLGRAPH (c_1, i, c_2, m), FORMALRET (m, x_2), ACTUALRET (i, x_1).
VFLOW (c_1, x_1, c_2, x_2)	VCALL (x_2, _, i), CALLGRAPH (c_2, i, c_1, m), THISVAR (m, x_1).

In Doop, new contexts are always introduced at a call-site, where variables and objects defined within the callee method acquire a new context by combining the caller context and the information associated with the caller instruction. As an example, for k-call-site sensitivity, a new context is created by appending the current call-site to the caller context (which is a list of call-sites), and deleting the oldest component (usually the first call-site in the current list) if the newly-formed context exceeds the predefined limit k. New contexts in a k-object-sensitive analysis are constructed in a similar way, with details given as follows.

We present an example of constructing new contexts at a call-site for a k-object-sensitivity analysis in Table IV, where two relations REACHABLE and CALLGRAPH are recursively defined. REACHABLE(c, m) denotes that method m is reachable with context c, and main, the entry method, is always reachable with the (pre-defined) empty context "[ ]". CALLGRAPH is used to extend a caller context to a callee context at a call-site. As the rule explains, given a call instruction i of the form "$\ldots = x.m'()$", if the context of the enclosing method (say m_0) of call-site i is c_1, and the matched method for the call-site is m (via LOOKUP), then m is reachable in context c_2. At the same time, we establish a CALLGRAPH relation from i in context c_1 to m in context c_2. Note that the semantics of context constructor NEWCTX depends on which context-sensitivity is applied. Since we assume k-object-sensitivity (i.e., a context is a list of heap objects), if the length of c_1 is less than k, then $c_2 = c_1 \circ o$, where "$\circ$" appends o to c_1; otherwise, c_2 is the list of removing the first object from $c_1 \circ o$.

B. Generating the One-Step Flow Relation

P/Taint only reports source-sink pairs in the form of LEAK$(c_1, value, c_2, invo)$. With this much information, it is often difficult for developers to find out how the taint source "$value$" in context c_1 is propagated to the sink location "$invo$" in context c_2, as the actual value flow may have passed through a number of inter-procedural calls and returns, as well as loading a value previously stored in the heap. In this section, we define new relations that help to make the flow of taint explicit. We extend the P/Taint system with the logic shown in Table V. To simplify the presentation, we assume k-object-sensitivity is applied, and the NEWCTX(o, c) constructor in Table V produces a new context by appending object o to context c and then removing the first component of c if the length of c is already k (so that after appending o, the new context does not have length exceeding k).

We define two new relations in this extension. ALIAS(c_1, v_1, c_2, v_2) can be straightforwardly derived from the available context-sensitive points-to analysis, in the sense that if two variables (probably in different contexts) may-points-to the same object, then they are in ALIAS relation. The construction of VFLOW relation is a bit more involved, as explained in the following three cases.

1) Variable $from$ flows to to in context c, if the enclosing method m of "$to = from$" is reachable in context c;
2) Given (c_1, v_1) and (c_2, v_2) in alias relation with both enclosing methods m_1 and m_2 reachable in context c_1 and c_2, respectively, and if there is a LOAD of v_1 via field f to variable x_1 and a STORE into v_2 via f from x_2, then this "load/store" can be paired to serve as a bridge which directs a value flow from x_2 to x_1;
3) The last three rules describe the inter-procedural value flow passing through a parameter, the return value, and this reference, respectively. Taking the last rule as an example, given a call-site from which an inter-procedural call (represented by the CALLGRAPH relation) can be established at call-site i from context c_2 to c_1 where the callee method is identified as m, we construct a one-step flow from the receiver x_2 (in context c_2) to this of m in context c_1.

C. Generating Traces for Taint Analysis

Formally, our analysis exports finite traces of the form $[w_0, w_1, w_2 \ldots w_k]$, where $w_i = (c_i, x_i)$ with value x_i in context c_i for all $1 \leq i \leq k-1$. A valid trace is required to satisfy the following three conditions.

1) Variable x_1 receives a return value from the pre-defined taint source v_0 in context $c_1 \in$ CTX.
2) Variable x_i in context c_i performs a one-step flow to variable x_{i+1} in context c_{i+1} for all $1 \leq i < k-1$.
3) Variable x_{k-1} passes value to a taint sink w_k.

TABLE VI
SAMPLE RULES FOR TAINT TRACE GENERATION

```
.type list = [next:list, x:ctype]
TTRACE (trace : list)          // a taint trace is a list of values
INTTRACE (trace : list)        // an intermediate trace
```

INTTRACE ([[nil, w_1], w_2])	CALLGRAPH (c, i, _, m), ACTUALRET(i, x), $w_2 = (c, x)$, $w_1 = (c, i)$, SOURCE (m, $type$).
INTTRACE ([t, w])	t = [t_1, w_1], $w_1 = (c_1, y)$, INTTRACE (t), $w = (c, x)$, VFLOW(c, x, c_1, y).
TTRACE ([t, w])	t = [t_1, w_1], $w_1 = (c, x)$, INTTRACE (t), SINK (m, n), CALLGRAPH (c, i, _ m), ACTUALARG (i, n, x), $w = (c, i)$.

The construction of each trace relies on the recursively typed records supported by the Datalog engine Soufflé [11]. For example, in order to encode a list of values, one needs to define a type list by [list, ctype] where ctype is the (component) type for the values in the list, and the recursion is terminated by a special predefined constant list nil. We present the definition of a taint trace (TTRACE) in Table VI. In this formulation, an intermediate trace (INTTRACE) encodes a list of component starting with a pre-defined source location (but it has not reached a sink location). The algorithm then tries to extend the list by looking for elements that are reachable from the last element in the existing list, following the VFLOW relation. Once a INTTRACE reaches a pre-defined taint sink, the search terminates with a complete TTRACE ready for output.

IV. IMPLEMENTATION AND EXPERIMENT

We have implemented our algorithm in Doop which extracts Datalog facts from the Shimple code which is an SSA-based Intermediate Representation (IR) in Soot. The experiment is conducted on a laptop equipped with an Intel Core® i5-8250U CPU@1.60GHz*8 and 16GB RAM, running UnionTech OS GNU/Linux 5.4.50-amd64-desktop. The software environment includes Doop (version 4.24.2), soufflé (version 2.0.2), graphviz (version 2.49.3), python (version 3.8.0) and OpenJDK (version 1.8.0). Our implementation is publicly available at https://github.com/lyj18688610256/LDoop.[3]

To measure the efficiency of our implementation, we carry out an experiment on Securibench Micro, an open source benchmark suite with 140 pairs of known source-to-sink flows located in a range of small web applications [13], from which we evaluate the precision, recall and efficiency of TTA. Different from the other methods that merely output ⟨source, sink⟩ pairs, we produce a set of graphically represented traces (with a context at each node of the trace) similar to what is shown in Fig. 2. We then start to manually check the traces in order to verify whether there are *false positives*. Fortunately, this procedure is not very time consuming, thanks to all taint flows being made explicit. We observe that sometimes our analysis produces a trace that contains a circle, i.e., $v_i = v_j$ with $0 < i < j < k$ where k is the length of the trace. One strategy that we currently use is to set up a search limit for traces, which rules out most of the redundant trace being reported. However, it should be noticed that some taint flows will be beyond our reach and becomes *false negatives* if the depth is too short. Currently, we are using an empirical limit for the Securibench Micro suite for an optimal result.

[3]A substantial amount of effort has been made to ensure that TTA aligns with the expected P/Taint analysis results. Due to recent updates in Doop, it is currently infeasible to directly reproduce the results as given in [8].

TABLE VII
ACCURACY OF TTA ON SECURIBENCH MICRO
(2-OBJECT-SENSITIVE+HEAP)

Suite	TP	FP	FN	Precision	Recall	F-score
Total	131	20	9	87%	94%	90%
aliasing	11	0	1	100%	92%	96%
arrays	10	3	1	77%	91%	83%
basic	61	0	0	100%	100%	100%
collections	16	3	0	84%	100%	91%
datastructures	6	0	0	100%	100%	100%
factories	3	0	0	100%	100%	100%
inter	11	6	5	65%	69%	67%
pred	3	5	0	38%	100%	55%
reflection	4	0	0	100%	100%	100%
sanitizers	2	0	2	100%	50%	67%
session	3	1	0	75%	100%	86%
strong_updates	1	2	0	33%	100%	50%

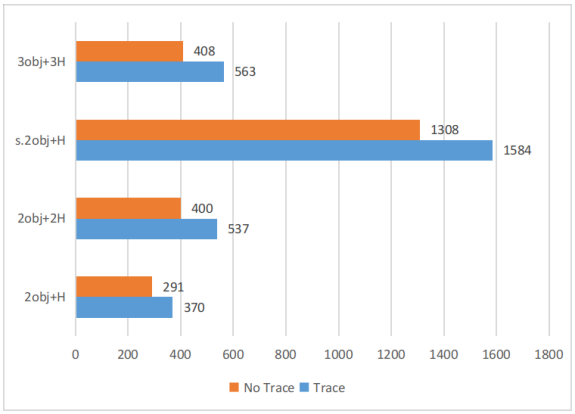

Fig. 4. Run-time (in seconds) with and without trace generation for different context-sensitivity options. For example, $3obj + 3H$ means that (lists of) at most 3 objects are used to represent method/variable contexts in CTX and also at most 3 objects are used to represent heap contexts in HCTX.

Accuracy of TTA: The accuracy results of TTA with the sensitive 2-object-1-heap analysis (i.e., $s.2obj + H$) on the benchmark suite are shown in Table VII, where **TP**, **FP** and **FN** represent *true positive*, *false positive* and *false negative*, respectively. From the table, one may find that for our implementation, the worst performed portion of the benchmark suite is the strong_updates folder (33% precision), which is due to that value flows for strong update usually require a *must* PTA, while the current TTA is based on *may* PTA. Nevertheless, over all, we still achieves 87% precision and 94% recall, with a 90% F_1 score. This effectively shows that TTP produces an acceptable result for the current benchmark suite.

Performance of TTA: We also check the run-time cost of our TTA implementation compared to the original P/Taint algorithm [8], with results shown in Fig. 4. When running the benchmark suite, for each context-sensitivity option, we ana-

lyze all programs in Securibench Micro simultaneously. From the table, one may find that our trace generation algorithm only adds a small amount of run-time overhead (less than 38%) to the original algorithm. Although the run-time cost of our implementation increases with larger contexts, for $2obj + H$ and $3obj + 3H$, the increased time cost becomes negligible. One possible explanation is that although $3obj + 3H$ analysis needs more time to compute variable contexts, it potentially produces less false positives. Therefore, the amount of time required for computing the traces becomes less.

V. Related Work

Perhaps the earliest taint analysis was implemented as the "taint mode" in the Perl language, which tracks taint flows via data-dependencies at runtime, i.e., taintness flows from the right hand side of each assignment to the left hand side, and reports an error if the arguments of a system call is tainted [4]. In general, a dynamic taint analysis tracks taint information at program execution, implemented with various methodologies such as binary instrumentation [16], [6], hardware-based taint tracking [23], and compiler-based taint tracking [27]. Static taint analysis heavily relies on underlying tools and representations of a program, including control flow graph [19], program dependence graph [10], program slicing [18], and type system [7]. In general, dynamic taint analysis provides security guarantee at runtime, while static analysis achieve better coverage at the cost of false positives. Our work is in the category of static taint analysis, with powerful context-sensitivity options from the underlying Doop framework and partial flow-sensitivity from the Shimple IR of Soot. Therefore, we are able to report potential vulnerabilities with relatively low false positive rate (c.f. Section IV). Moreover, our focus is on producing explicit taint traces for software developers.

Our work is an extension on P/Taint [8], a unified points-to analysis and taint analysis in the Doop framework [2] equipped with a range of context-sensitivity options. Given that P/Taint only reports source-sink pairs which are hard to comprehend by human, we explore ways of trace generation from the explicit flow relation in Type Flow Analysis [28], which express a flow relation by joining intraprocedural flow, inter-procedural flow, and pairing of load and store. Such a one-step relation can be connected with a recursive encoding technique supported in the Datalog engine Soufflé [11].

VI. Conclusion and Future Work

We have introduced an extension for context-sensitive taint analysis for Java (P/Taint), called Taint Trace Analysis (TTA), which produces a set of taint traces with context information included. Little run-time overhead on top of P/Taint has been shown for our implementation. In contrast to most existing taint analysis works, the produced taint traces from TTA may provide more useful information for the detection and tracking of security vulnerabilities in Java web applications. In the future, we will extend the existing work to analyze Android apps, and we also plan to further optimize our algorithm to achieve higher precision and a quicker visualization procedure.

References

[1] L. O. Andersen. *Program analysis and specialization for the C programming language*. PhD thesis, DIKU, University of Copenhagen, May 1994.

[2] M. Bravenboer and Y. Smaragdakis. Strictly declarative specification of sophisticated points-to analyses. In *OOPSLA'09*, page 243–262, 2009. https://bitbucket.org/yanniss/doop/.

[3] T. Chen and S. Abu-Nimeh. Lessons from stuxnet. *IEEE Computer*, 44(4):91–93, 2011.

[4] T. Christiansen. Perl security. https://perldoc.perl.org/. Taint module available November 1997.

[5] C. Cifuentes, N. Keynes, L., N. Hawes, and M. Valdiviezo. Transitioning parfait into a development tool. *IEEE Security and Privacy*, 10(3):16–23, 2012.

[6] J. Clause, W. Li, and A. Orso. Dytan: A generic dynamic taint analysis framework. In *Proceedings of the 2007 International Symposium on Software Testing*, pages 196—-206, 2007.

[7] J. S. Foster, T. Terauchi, and A. Aiken. Flow-sensitive type qualifiers. In *PLDI'02*, pages 1—-12, 2002.

[8] N. Grech and Y. Smaragdakis. P/Taint: unified points-to and taint analysis. In *OOPSLA'17*, pages 102:1–102:26, 2017.

[9] B. Greenman. Datalog for static analysis. History of programming language seminar (2017) https://github.com/nuprl/hopl-s2017.

[10] C. Hammer, J. Krinke, and G. Snelting. Information flow control for java based on path conditions in dependence graphs. In *IEEE International Symposium on Secure Software Engineering*, 2006.

[11] H. Jordan, B. Scholz, and P. Subotić. Soufflé: On synthesis of program analyzers. In *CAV'16*, pages 422–430, 2016.

[12] G. Kastrinis and Y. Smaragdakis. Hybrid context-sensitivity for points-to analysis. In *PLDI'13*, pages 423–434, 2013.

[13] B. Livshits. *Improving Software Security with Precise Static and Runtime Analysis*. PhD thesis, Stanford University, 2006.

[14] A. Milanova, A. Rountev, and B. G. Ryder. Parameterized object sensitivity for points-to analysis for Java. *ACM Transations on Software Engineering and Methodology*, 14(1):1–41, 2005.

[15] D. Moore, V. Paxson, S. Savage, C. Shannon, S. Staniford, and N. Weaver. Inside the slammer worm. *IEEE Security and Privacy*, 1(4):33–39, 2003.

[16] J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *NDSS'05*, 2005.

[17] OWASP. Top 10 web application security risks. https://owasp.org/www-project-top-ten/. Accessed: 2021-11-23.

[18] M. Pistoia, L. Koved R. J. Flynn, and V. C. Sreedhar. Interprocedural analysis for privileged code placement and tainted variable detection. In *ECOOP'05*, pages 362—-386, 2005.

[19] B. Scholz, C. Zhang, and C. Cifuentes. User-input dependence analysis via graph reachability. In *SCAM'08*, pages 25—-34, 2008.

[20] O. G. Shivers. *Control-flow Analysis of Higher-order Languages or Taming Lambda*. PhD thesis, Carnegie Mellon University, 1991. UMI Order No. GAX91-26964.

[21] Y. Smaragdakis, M. Bravenboer, and O. Lhoták. Pick your contexts well: understanding object-sensitivity. In *POPL'11*, pages 17–30, 2011.

[22] M. Sridharan, S. Artzi, M. Pistoia, S. Guarnieri, O. Tripp, and R. Berg. F4F: taint analysis of framework-based web applications. In *OOPSLA 2011*, pages 1053–1068. ACM, 2011.

[23] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas. Secure program execution via dynamic information flow tracking. In *ASPLOS'04*, page 85–96, 2004.

[24] Symantec. Top 10 web application security risks. Internet Security Threat Report, Vol. 21., April 2016.

[25] O. Tripp, M. Pistoia, S. J. Fink, M. Sridharan, and O. Weisman. TAJ: effective taint analysis of web applications. In *PLDI'09*, pages 87–97. ACM, 2009.

[26] US-CERT. Vulnerability note vu#881872, sun solaris telnet authentication bypass vulnerability. http://www.kb.cert.org/vuls/id/881872. Accessed: 2021-11-23.

[27] W. Xu, S. Bhatkar, and R. Sekar. Taint-enhanced policy enforcement: A practical approach to defeat a wide range of attacks. In *Proceedings of the 15th USENIX Security Symposium*, page 121–136, 2006.

[28] X. Zhuo and C. Zhang. TFA: an efficient and precise virtual method call resolution for Java. *Formal Aspects of Computing*, 32:395—416, 2020.