Context-Sensitive Staged Static Taint Analysis for C using LLVM

Xavier NOUMBISSI NOUNDOU

University of Bremen, Germany

Abstract. A large amount of software security vulnerability exploits vulnerabilities such as buffer overflows, SQL injections, cross-site scripting attacks and others; these are typically caused by data flowing from untrusted program input sources into sensible program functions. We define a tainted path as a program execution path from an untrusted program input source into a sensible program location. This paper presents a static taint analysis that computes tainted paths in C programs and that doesnt require any program annotations. Our static taint analysis algorithm is built upon the iterative dataflow framework [12][18] and has been implemented in the tool SAINT (Simple Static Taint Analysis Tool). Our static taint analysis is interprocedural, flow-sensitive, and developers can choose to run it either with context-sensitivity or without. We have implemented our analysis using the LLVM compiler infrastructure.

1 Introduction

Software vulnerabilities are security threats that exist in an application. Software vulnerabilities allow malovelent users to exercise unauthorized control of the application through supplied input. There are several kinds of software vulnerabilities: buffer overflows, format string attacks, SQL injection, etc. Researchers have worked on dynamic [6] [11][7], static [23] [11] [4] [5] [10] [16] [2] [21], and hybrid techniques [24] to find security vulnerabilities in software.

This paper introduces the concept of *tainted path.* A tainted path is a program execution path from a program input source into a sensible program location. A tainted path represent a software vulnerability. This paper presents a static taint analysis that computes tainted data and tainted paths in C programs. Our implementation of the taint analysis uses the LLVM framework [13], and does not require user annotations. Our static taint analysis is flow-sensitive, interprocedural, and developers can choose to run it with context-sensitivity or without. In taint analysis, a *source* is a program location that allows a value from the environment into the program. This may occur through the return value of a system call, user input, etc. A value from the program environment that has not been *validated* and *sanitized* is called a *tainted value.* A *sink* is a sensible (vulnerable) function.

Data validation is the process of checking that data has the expected form. For instance, checking that a string input has the format of an email address. *Data sanitization* is the process of checking that validated data is safe in a particular context. For instance, escaping string input before using it in a SQL query. A function that sanitizes application's external input is called a *sanitizer*. Once a value has been sanitized, it is tagged as *not tainted*. In the following, we assume that sanitized data has been validated.

Static taint analysis searches for tainted values and warn developers for each tainted value so they can validate and sanitize the tainted value to avoid software vulnerability exploits at runtime. Taint analysis proceeds by first tagging values from sources as tainted. Once tagged, the tainted values are propagated through the entire program.

Taint propagation is the process of marking values as *tainted* if they result from an operation that involved tainted data. This can be an arithmetic operation (addition, multiplication, etc.), a program assignment or other type of program instructions. Finally, a taint analysis emits a warning whenever a tainted value is used at a sink location. Taint propagation can be *data-flow* or *control-flow* based. Data-flow based taint propagation exists due to data dependencies in the program (e.g. assigning the value of tainted variable s_u to another variable s_d). Control-flow based taint propagation is due to control dependencies (e.g. if tainted variable s_t is used in a branch condition, values from program instructions inside that branch become tainted.). Data-flow based taint propagation is also called *explicit taint propagation*, and control-flow based taint propagation is called *implicit taint propagation*. Our static taint analysis searches for *tainted paths* and implements both control-flow and data-flow based taint propagation.

This paper makes the following contributions:

- It defines the concept of *tainted path*, which is a program execution path from a taint source to a taint sink.
- It describes SAINT, a whole-program static taint analysis that is flow-sensitive, interprocedural and context-sensitive. SAINT computes tainted paths in C programs and is available for download at `https://github.com/xaviernoumbis/saint`. To the best of our knowledge, SAINT is the only tool that implements static taint analysis for a static language using the iterative dataflow framework [12] [18]. Please look at Table 4 in Section 5.

The paper is organized as follows: Section 2 introduces the concept of tainted path, and Section 3 gives an overview of the LLVM intermediate representation. Section 4 presents the taint analysis algorithm, and Section 5 presents experimental results. Finally, Section 6 discusses related work and Section 7 concludes.

2 Tainted Paths

In this section we introduce the term *tainted path*. We define a *tainted path* as a program execution path from a taint source to a taint sink. Let us consider the three-address code in Figure 1. We represent a line of code with Lx where L means line and x represents a line number. The program path $<$ L2, L3, L5, L6, L8, L9 $>$ defines a tainted path while program paths $<$ L2, L3, L5, L6, L7, L11, L4 $>$ and $<$ L2, L3, L4 $>$ don't define tainted paths.

```
     int calculate(int);
     void mysql_taint(int); //taint sink

     int main() {
L1:    int x;
L2:    scanf(%d, &x); //taint source
L3:    int y = calculate(x);
L4:    return 0;
     }

     int calculate(int x) {
L5:    int sum = 0;
L6:    if (sum >= x)
L7:      goto L11;
L8:    sum = sum + 1;
L9:    mysql_taint(sum);
L10:   goto L6;
L11:   return sum;
     }
```

Fig. 1: Code example in three-address form

3 LLVM

Abstract intructions	**Description**	**Code in C**	**Formal description**
ALLOC	Memory allocation	$v = malloc(...)$	$v \in \mathcal{T}$
COPY	Copy instruction	$p = q$	$p, q \in \mathcal{T}$
LOAD	Load instruction	$p = *q$	$p \in \mathcal{T}, q \in \mathcal{A}$
STORE	Store instruction	$*p = q$	$p \in \mathcal{A}, q \in \mathcal{T}$
CALL	Call instruction	$r = call$ func (p)	$r \in \mathcal{T}$

Table 1: LLVM abstract instructions types. In LLVM intermediate representation, $\mathcal{A}$ and $\mathcal{T}$ represent respectively the set of address-taken variables and the set of top-level variables.

This section gives an overview of LLVM[1] (*Low Level Virtual Machine*) and its intermediate representation (IR), which we use as basis for the description of our taint analysis. LLVM is a compiler framework [13] that contains several components and libraries that help developers in building compilers and compiler tools (e.g. static analyses, etc.). LLVM primarily processes source code written in C, C++, and Objective C. LLVM libraries are written in C++. Table 1 shows

[1] http://llvm.org

the abstract instruction types we consider in the LLVM IR for our analysis. In the following, we present the LLVM intermediate representation. Our presentation is based on the descriptions given by Hardekopf et al. [9] and by Lhoták et al. [15]. LLVM's IR uses partial static single assignment (partial SSA) and assumes the existence of two types of variables in C code: *top-level* variables and *address-taken* variables.

3.1 Top-level variables

Top-level variables are variables that are never accessed via a pointer in the program code. LLVM converts top-level variables into SSA form when building the LLVM IR. The memory address of top-level variables is never copied to another variable (i.e. they are never applied the address-of operator (&) in the C programming language). In the LLVM IR, top-level variables are only accessed using **ALLOC** and **COPY** instructions. This paper denotes the set of top-level variables with $\mathcal{T}$. In Figure 1 y is a top-level variables ($y \in \mathcal{T}$).

3.2 Address-taken variables

Address-taken variables are never accessed directly through their first declared name. Address-taken variables are only accessed indirectly with pointer variables and **LOAD** and **STORE** instructions. In fact, address-taken variables are those ones on which the address-of operator (`&`) was applied. This paper uses $\mathcal{A}$ for the set of all address-taken variables. Variable `x` in Figure 1 is for instance an address-taken variable ($x \in \mathcal{A}$).

4 Staged Static Taint Analysis

Our taint analysis is interprocedural and runs either context-insensitively or context-sensitively. Any form of the interprocedural analysis is always preceded by an intraprocedural analysis that computes initial taint information that is reused by the interprocedural analyses. The intraprocedural analysis detects taint sources and initializes a summary table which contains taint information about program functions' formal parameters and return value. The use of a summary table allows fast access to key information about program procedures. This is especially useful during the subsequent interprocedural phases. Table 2 shows the Gen-set for the abstract program statements **ALLOC**, **COPY**, **LOAD**, **STORE**, and **CALL**. Algorithm 3 shows the transfer functions for the different types of program statements.

4.1 Taint sources and taint sinks

Program statements that initially taint variables (*taint sources*) are discovered during the intraprocedural analysis, described later in this section. The analysis handles per default a subset of the C standard library as taint sources: `getc`,

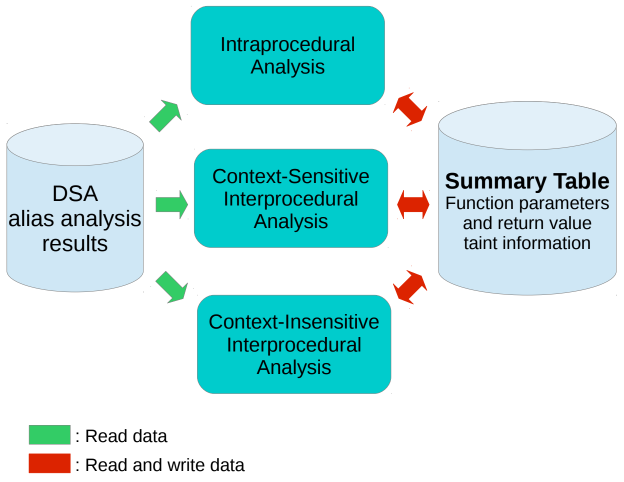
Intraprocedural Analysis
DSA alias analysis results
Context-Sensitive Interprocedural Analysis
Summary Table
Function parameters and return value taint information
Context-Insensitive Interprocedural Analysis
🟩 : Read data
🟥 : Read and write data

Fig. 2: SAINT analysis architecture

Abstract instructions	**Code in C**	**Gen-sets**
ALLOC	s: $v = malloc(...)$	$\emptyset$
COPY	s: $p = q$	$\{p\}$ iff $q \in \text{IN}[s]$
LOAD	s: $p = *q$	$\{t_j \| t_j = toplevel(a_j) \wedge a_j \in points\_to_{[\bar{s}]}(q)$ $\wedge t_j \in \text{IN}[s]\}$
STORE	s: $*p = q$	$\{t_j \| t_j = toplevel(a_j) \wedge a_j \in points\_to_{[\bar{s}]}(p)\}$ iff $q \in \text{IN}[s]$

Table 2: Gen-sets for the abstract instructions **ALLOC**, **COPY**, **LOAD**, and **STORE**. Kill-sets are empty.

`scanf`, `gets`, `fopen`, etc. Vulnerable functions that use tainted variables (*taint sinks*) are gradually discovered during the various phases of the analysis. SAINT has a configuration file where developers can register additional taint source and taint sink functions.

4.2 Taint propagation

SAINT performs *explicit* and *implicit* taint propagation (data- and control-flow taint propagation). Explicit taint propagation tracks variables that are tainted due to assignment statements. In Figure 1 the assignment in line L3 is an instance of explicit taint propagation. Variable y becomes tainted since it gets assigned the return value of `compute`, which may be a tainted value.

4.3 Formalisms

This paper uses the following elements to describe the taint analysis as an instance of the iterative dataflow analysis framework [18]: *Var* is the set of all program variables, $Proc$ is the set of all program functions and procedures[2], $Inst$ is the set of all program statements, $formals : Proc \rightarrow 2^{Var}$ returns the set of formal parameters of a function, $toplevel : \mathcal{A} \rightarrow \mathcal{T}$ returns the top level variable of an address-taken variable, $taint : Proc \times Integer \rightarrow bool$ returns `true` if function f always taints its k^{th} formal parameter, and $aliases : Var \rightarrow 2^{Var}$ returns the alias set of a variable as returned by the pointer analysis.

The analysis dataflow set is $Inst \times 2^{Var}$. 2^{Var}, the powerset of all program variables (Var) is by definition a lattice. In effect, $Inst \times 2^{Var}$ is a mapping from the set of all program instructions $Inst$ into 2^{Var}. $Inst \times 2^{Var}$ is therefore a lattice by definition[3]

At a statement labelled s, the incoming dataflow set $IN[s]$ is the set of all program variables that are tainted before statement s. If a variable v is not tainted before statement s, then $v \notin IN[s]$; otherwise $IN[s]$ contains v.

The execution of the transfer function at a statement labelled s eventually discover new tainted variables. $OUT[s]$ describes the new set of tainted variables after statement s.

The function $sumTable : Proc \times Integer \rightarrow \{True, False\}$ describes the summary table, and reveals for a procedure $f \in Proc$ and its k^{th} formal parameter whether the k^{th} formal parameter is tainted or not. For instance $sumTable[f][2] = True$ means that the third formal parameter of function f is tainted.

We use ret to represent the return value of a function f. ret is tainted implies that $sumTable[f][ret] = True$, otherwise $sumTable[f][ret] = False$. $points\_to_{[\overline{s}]}(q)$ describes the set of aliases of variable q before statement s.
$read\_taint\_arg : Proc \rightarrow Integer$ reads

4.4 Handling of pointers

Our taint analysis is designed to work using the results of a previously computed pointer analysis. SAINT uses the pointer analysis `DSA` (`Data Structure Analysis`) [14] which is implemented in the tool `poolalloc` [4]. `DSA` is a field- and context-sensitive pointer analysis. `DSA` uses full heap cloning (by acyclic call paths) and scales well with programs in a size range of 100K-200K lines of C code. $points\_to_{[\overline{s}]}(q)$ describes the set of aliases of variable q before statement s. Our taint analysis uses the results of a pointer analysis via the function $points\_to$ to update dataflow sets.

4.5 Intraprocedural analysis

The intraprocedural analysis always runs first before any interprocedural analysis, and is responsible for discovering taint sources. During the intraprocedural analysis, program functions are analyzed in the reverse topological order of the

[2] We will use the terms function and procedure interchangeably in the remainder of this paper
[3] Please consult the axiomatic propositions that define lattices
[4] https://github.com/poolalloc

```
input  : caller : $Proc$, s : $Inst$, $k$ : $Integer$
output:
switch TypeOf(s) do
    case CALL [r = $call$ source($a_0, a_1, ..., a_n$)]
        k := read_taint_arg(source)
        foreach $v_j \in points\_to_{[\bar{s}]}(a_k)$ do
            if $v_j \in$ IN[s] then
                OUT[s] := OUT[s] $\cup \{v_j\}$
            end
        end
    end
endsw
foreach $f_k \in$ formals(caller) do
    if IN[s] $\neq$ OUT[s] and $f_k \in$ OUT[s] then
        sumTable[caller][k] := $True$
    end
end
```

Algorithm 1: Intraprocedural analysis' transfer function for `CALL` statements.

call graph (i.e starting from the leaves of the callgraph). The analysis works on each function body and do not take into account interprocedural control flow. The computed data flow sets are reused later by the subsequent interprocedural analyses. In particular, taint information about function formal parameters and return value is kept in a summary table. All variables tainted due to source functions are found during the intraprocedural analysis. In Figure 1 for instance, the intraprocedural analysis detects that variable `sum` in function `compute` may be tainted at line L2 due to the call to `scanf`, which the analysis considers as a taint source. Algorithm 1 shows the transfer function for function call statements, which handles the discovery of taint sources during the intraprocedural analysis. Transfer functions for all other statement types are the same as the ones in Algorithm 3.

4.6 Context-insensitive analysis

The context-insensitive analysis algorithm performs in the topological order of the call graph (i.e. the algorithm runs from the program entry point to the leaves of the call graph). The context-insensitive analysis only uses taint assumptions from the summary table to update data flow sets at program points. For instance, the intraprocedural analysis of function `main` marks the return value of `compute` (stored in variable `sum`) as tainted in the summary table. At line 6 in `main`, the context-insensitive analysis would take into account that the return value of `compute` is stored into variable `y`. Thus, `y` becomes a tainted variable during the context-insensitive analysis. Algorithm 2 in the following illustrates the context-insensitive flow function for `CALL` statements.

```
input  : caller : $Proc$, s : $Inst$
output:
switch TypeOf(s) do
    case CALL [r = $call$ func($a_0, a_1, ..., a_n$)]
        if $True$ = sumTable[func][$ret$] then
            foreach $\mathsf{v}_j \in points\_to_{[\bar{s}]}$(r) do
                if $\mathsf{v}_j \in$ IN[s] then
                    OUT[s] := OUT[s] $\cup$ {$\mathsf{v}_j$}
                end
            end
        end
        foreach k $\in \{0, 1, ..., n\}$ do
            if $True$ = sumTable[func][k] then
                foreach $\mathsf{v}_j \in points\_to_{[\bar{s}]}$($\mathsf{a}_k$) do
                    if $\mathsf{v}_j \in$ IN[s] then
                        OUT[s] := OUT[s] $\cup$ {$\mathsf{v}_j$}
                    end
                end
            end
        end
    end
endsw
```

Algorithm 2: Context-insentive interprocedural transfer function for `CALL` statements

4.7 Context-sensitive analysis

The context-sensitive analysis algorithm works in the topological order of the call graph, and uses information from the summary table that were produced by previous analyses. Even if the context-insensitive analysis was run before, the context-sensitive analysis eventually writes more precise information in the summary table. At call sites, the context-sensitive analysis propagates actual parameters taint information from the caller into the callee. At callee exits, newly computed taint information are propagated back from the callee context to the caller context. Algorithm 3 illustrates context-sensitive transfer functions for all statement types.

The context-sensitive algorithm implements the interprocedural *call-string approach* by Sharir and Pnueli [22]. The call-string length in SAINT implementation is 2, which means that 2 is the depth at which context-sensitive calls are analyzed.

5 Experimental Evaluation

We ran the analysis on an `Intel P8400 @ 2.26 GHz` with 2 cores and 2 GB of RAM, running `Linux`. Table 3 shows the analysis results of the taint analysis,

```
input  : caller : $Proc$, s : $Inst$, k : $\{1, 2, ..., n\}$, cnfMax : $\{1, 2, ..., n\}$
output:
switch TypeOf(s) do
    case COPY [p = q]
        if q $\in$ IN[s] then
            OUT[s] := OUT[s] $\cup$ {p}
        end
    end
    case LOAD [$p = *q$]
        foreach $a_j \in points\_to_{[\bar{s}]}(q)$ do
            $t_j$ := toplevel($a_j$)
            if $t_j \in$ IN[s] then
                OUT[s] := OUT[s] $\cup \{t_j\}$
            end
        end
    end
    case STORE [$*p = q$]
        if q $\in$ IN[s] then
            foreach $a_j \in points\_to_{[\bar{s}]}(p)$ do
                $t_j$ := toplevel($a_j$)
                if $t_j \in$ IN[s] then
                    OUT[s] := OUT[s] $\cup \{t_j\}$
                end
            end
        end
    end
    case CALL [r = $call$ func($a_0, a_1, ..., a_n$)]
        if caller $\neq$ func then
            foreach $f_j \in$ formals(func) do
                if $False$ = summary[func][$j$] then
                    foreach $v_j \in points\_to_{[\bar{s}]}(a_j)$ do
                        if $v_j \in$ IN[s] then
                            OUT[s] := OUT[s] $\cup \{v_j\}$
                        end
                    end
                end
            end
            if k < cnfMax then
                k := k + 1
                csInterFlow(caller, func, k, cnfMax)
            end
            foreach $f_j \in$ formals(func) do
                if $False$ = summary[func][$j$] $and$ OUT[$f_j$] $\neq$ IN[$f_j$] then
                    summary[func][$j$] := $True$
                    foreach $v_j \in points\_to_{[\bar{s}]}(a_j)$ do
                        if $v_j \in$ IN[s] then
                            OUT[s] := OUT[s] $\cup \{v_j\}$
                        end
                    end
                end
            end
        end
    end
endsw
```
Algorithm 3: csInterFlow: Context-sentive analysis transfer functions

Program (version)	**#SLOC**	**#Tainted paths**	**Tainted values (%)**	**Time (seconds)**
`mongoose` (4.1)	$4k$			$2.14s$
`vlc-input` (2.1.2)	$16k$			$0.03s$
`openssl-ssl` ($1.0.1f$)	$40k$			$0.44s$
`apache` (2.4.7)	$144k$			n/a

Table 3: Saint's taint analysis experiment results

ran in the following order: intraprocedural analysis, context-sensitive analysis, and context-insensitive analysis.

6 Related Work

Tools	**Language**	**Technique**	**Availability**	**Appearance Year**
parfait [19]	C, C++	graph reachability algorithm	Commercial	2008
coverity	C, C++, Java	?	Commercial	?
fortify	C, C++, Java	?	Commercial	?
veracode	C, C++, Java	?	Commercial	?
taj [23]	Java	Program slicing	Commercial	2009
pixy [10]	PHP	iterative dataflow framework	Research	2006
FlowDroid [1]	Java (Android)	IFDS framework	Research	2014
cqual [21]	C	Type system	Research	2001
stac [3]	C	Type system	Research	2009
$*$ **saint**	C	iterative dataflow framework	Research	2015

Table 4: Static taint analysis tools for security vulnerability search

There has been a lot of work in the area of taint analysis and its applications. This section discusses static taint analysis-based projects.

Parfait from Oracle Labs checks for bugs in C programs [5]. `Parfait` is built on top of LLVM and uses a demand driven analysis to mitigate scalability issues inherent to standard forward dataflow analysis techniques. `Parfait` does not require annotations from developers and is advertised to scale to millions of lines of code. For security vulnerabilities, `Parfait` implements a taint analysis [20] as pre-processing filter that is linear in the number of statements and dependencies. `Parfait`'s taint analysis is formulated as a graph reachability problem.

`Parfait` implements both a context-insensitive and a context-sensitive solution for its taint-analysis, and adds a may-function alias analysis to LLVM to better support the accuracy of the taint analysis. Similarly to `Parfait`, SAINT may also perform either a context-insensitive or a context-sensitive analysis. SAINT is also based on LLVM, and do not require any annotations from developers. It is not possible to use a third party alias analysis with `Parfait`. In contrast, SAINT users have the possibility to change the pointer analysis library it uses.

Pixy is a tool that statically scans for cross-site scripting vulnerabilities in PHP scripts [10]. `Pixy` implements a flow-sensitive, context-sensitive dataflow analysis. `Pixy` also creates and uses an alias and literal analysis for PHP.

Livshits et al. present a tool for finding security vulnerabilities in web applications written in Java [16]. The tool is based on `bddbddb` [5], which automatically generates context-sensitive program analyses for specifications written in `Datalog` [25]. `bddbddb` uses binary decisions diagrams to represent and manipulate points-to analysis results for different contexts in a Java program. The authors implement taint propagation on top of the points-to analysis results generated by `bddbddb`. In effect, developers specify vulnerabilities in the `PQL` language [17], which is a syntactic sugar for `Datalog`. That is, each vulnerability specification corresponds to a set of `PQL` queries.

CQual by Shankar et al. detects format string vulnerabilities in C programs [21]. `CQual`'s analysis is based on type qualifiers[8] and type inference. Given a C program, an initial subset of the program is annotated with the type qualifiers `tainted` and `untainted`. `CQual` then uses a set of inference rules to generate type (qualifier) constraints over the program. Pointer reasoning is done via some rules of the type system. The analysis then warns the user of a format string vulnerability whenever the program does not type checks due to an unsatisfiable constraint. According to results published in [21], `CQual` performs on average 85s for a code base of 20k lines of code, and takes 268s to analyze the largest program of 43k. In contrast to `CQual`, users of SAINT may experiment with different pointer analysis libraries and choose among them. SAINT also do not require any annotations.

7 Conclusions

In this paper, we have presented SAINT, a whole-program static taint analysis that is flow-sensitive, interprocedural and that can be run either with context-sensitivity or without. SAINT computes tainted paths in C programs and is available for download at `https://github.com/xaviernoumbis/saint`. SAINT's taint analysis is sound. We plan to implement transfer functions for sanitizer functions in future realeases of SAINT. To the best of our knowledge SAINT

[5] http://suif.stanford.edu/bddbddb

is the only tool that implements static taint analysis for a static language using the iterative dataflow framework [12][18]. The staged nature of SAINT makes it suitable for integration in integrated development environments (e.g: Eclipse, Anjuta, etc.).

Acknowledgments

We would like to thank Jan Peleska for his moral, academic, and financial support to our research. We also thank Karsten Sohr for his review comments.

References

1. Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 259–269, New York, NY, USA, 2014. ACM.
2. Dzintars Avots, Michael Dalton, V. Benjamin Livshits, and Monica S. Lam. Improving software security with a c pointer analysis. In *Proceedings of the 27th international conference on Software engineering*, ICSE '05, pages 332–341, New York, NY, USA, 2005. ACM.
3. Dumitru CEARA. Detecting software vulnerabilities static taint analysis. http://tanalysis.googlecode.com/files/DumitruCeara_BSc.pdf, 2009. Internship Report at Verimag Laboratory.
4. Richard Chang, Guofei Jiang, Franjo Ivancic, Sriram Sankaranarayanan, and Vitaly Shmatikov. Inputs of coma: Static detection of denial-of-service vulnerabilities. In *Proceedings of the 2009 22nd IEEE Computer Security Foundations Symposium*, CSF '09, pages 186–199, Washington, DC, USA, 2009. IEEE Computer Society.
5. Cristina Cifuentes and Bernhard Scholz. Parfait: designing a scalable bug checker. In *SAW '08: Proceedings of the 2008 workshop on Static analysis*, pages 4–11, New York, NY, USA, 2008. ACM.
6. James Clause, Wanchun Li, and Alessandro Orso. Dytan: a generic dynamic taint analysis framework. In *Proceedings of the 2007 international symposium on Software testing and analysis*, ISSTA '07, pages 196–206, New York, NY, USA, 2007. ACM.
7. William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 1–6, Berkeley, CA, USA, 2010. USENIX Association.
8. Jeffrey S. Foster, Manuel Fähndrich, and Alexander Aiken. A theory of type qualifiers. In *Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation*, PLDI '99, pages 192–203, New York, NY, USA, 1999. ACM.
9. Ben Hardekopf and Calvin Lin. Flow-sensitive pointer analysis for millions of lines of code. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '11, pages 289–298, Washington, DC, USA, 2011. IEEE Computer Society.

10. Nenad Jovanovic, Christopher Kruegel, and Engin Kirda. Pixy: A static analysis tool for detecting web application vulnerabilities (short paper). In *Proceedings of the 2006 IEEE Symposium on Security and Privacy*, SP '06, pages 258–263, Washington, DC, USA, 2006. IEEE Computer Society.
11. Adam Kieżun, Philip J. Guo, Karthick Jayaraman, and Michael D. Ernst. Automatic creation of SQL injection and cross-site scripting attacks. In *ICSE'09, Proceedings of the 31st International Conference on Software Engineering*, pages 199–209, Vancouver, BC, Canada, May 20–22, 2009.
12. Gary A. Kildall. A unified approach to global program optimization. In *Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '73, pages 194–206, New York, NY, USA, 1973. ACM.
13. Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, CGO '04, pages 75–, Washington, DC, USA, 2004. IEEE Computer Society.
14. Chris Lattner, Andrew Lenharth, and Vikram Adve. Making Context-Sensitive Points-to Analysis with Heap Cloning Practical For The Real World. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'07)*, San Diego, California, June 2007.
15. Ondrej Lhoták and Kwok-Chiang Andrew Chung. Points-to analysis with efficient strong updates. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '11, pages 3–16, New York, NY, USA, 2011. ACM.
16. V. Benjamin Livshits and Monica S. Lam. Finding security errors in Java programs with static analysis. In *Proceedings of the 14th Usenix Security Symposium*, pages 271–286, August 2005.
17. Michael Martin, Benjamin Livshits, and Monica S. Lam. Finding application errors and security flaws using pql: A program query language. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '05, pages 365–383, New York, NY, USA, 2005. ACM.
18. Steven S. Muchnick. *Advanced Compiler Design and Implementation.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997.
19. Bernhard Scholz, Chenyi Zhang, and Cristina Cifuentes. User-input dependence analysis via graph reachability. Technical report, Mountain View, CA, USA, 2008.
20. Bernhard Scholz, Chenyi Zhang, and Cristina Cifuentes. User-input dependence analysis via graph reachability. Technical report, Mountain View, CA, USA, 2008.
21. Umesh Shankar, Kunal Talwar, Jeffrey S. Foster, and David Wagner. Detecting format string vulnerabilities with type qualifiers. In *Proceedings of the 10th conference on USENIX Security Symposium - Volume 10*, SSYM'01, pages 16–16, Berkeley, CA, USA, 2001. USENIX Association.
22. Micha Sharir and Amir Pnueli. *Two approaches to interprocedural data flow analysis*, chapter 7, pages 189–234. Prentice-Hall, Englewood Cliffs, NJ, 1981.
23. Omer Tripp, Marco Pistoia, Stephen J. Fink, Manu Sridharan, and Omri Weisman. Taj: effective taint analysis of web applications. *SIGPLAN Not.*, 44(6):87–97, June 2009.
24. Omer Tripp and Omri Weisman. Hybrid analysis for javascript security assessment. New York, NY, USA, 2011. ACM Conference on the Foundations of Software Engineering (ESEC/FSE '11 – Industrial Track).

25. Jeffrey D. Ullman. *Principles of Database and Knowledge-base Systems, Vol. I.* Computer Science Press, Inc., New York, NY, USA, 1988.