Efficient static checker for tainted variable attacks

Andrei Rimsa, Marcelo d'Amorim, Fernando Magno Quintão Pereira, Roberto S. Bigonha

PII: S0167-6423(13)00073-7
DOI: http://dx.doi.org/10.1016/j.scico.2013.03.012
Reference: SCICO 1532

To appear in: *Science of Computer Programming*

Received date: 15 December 2011
Revised date: 9 October 2012
Accepted date: 23 March 2013

Efficient Static Checker for Tainted Variable Attacks

Andrei Rimsa[a], Marcelo d'Amorim[b], Fernando Magno Quintão Pereira[a], Roberto S. Bigonha[a]

[a] *UFMG – Av. Antônio Carlos 6627, 31.270-010, Belo Horizonte, Brazil*
[b] *UFPE – Av. Jornalista Aníbal Fernandes S/N, 50.740-560, Cid. Universitária, Recife, Brazil*

Abstract

Tainted flow attacks originate from program inputs maliciously crafted to exploit software vulnerabilities. These attacks are common in server-side scripting languages, such as PHP. In 1997, Ørbæk and Palsberg formalized the problem of detecting these exploits as an instance of type-checking, and gave an $O(V^3)$ algorithm to solve it, where V is the number of program variables. A similar algorithm was, ten years later, implemented on the Pixy tool. In this paper we give an $O(V^2)$ solution to the same problem. Our solution uses Bodik *et al.*'s extended Static Single Assignment (e-SSA) program representation. The e-SSA form can be efficiently computed and it enables us to solve the problem via a sparse dataflow analysis. Using the same infrastructure, we compared a state-of-the-art dataflow solution with our technique. Both approaches have detected 36 vulnerabilities in well known PHP programs. Our results show that our approach tends to outperform the dataflow algorithm for larger inputs. We have reported the new bugs that we found, and an implementation of our algorithm is publicly available at https://github.com/rimsa/tainted-phc.git.

Key words: Tainted flow analysis, security vulnerability, static analysis.

Email addresses: rimsa@live.com (Andrei Rimsa), damorim@cin.ufpe.br (Marcelo d'Amorim), fernando@dcc.ufmg.br (Fernando Magno Quintão Pereira), bigonha@dcc.ufmg.br (Roberto S. Bigonha)

1. Introduction

Web applications, so pervasive in the Internet, often manipulate sensitive information. It comes to no surprise that these applications are usual targets of *cyber attacks* [1]. These attacks typically initiate with a remote individual, the attacker, carefully forging inputs to corrupt a running system.

An important class of cyber-attacks is known as *Tainted Flow Attacks*. It manifests when a remote individual explores potential leaks in the system via its public interface. For the most popular cases, the interface is a web service and the vulnerability is the lack of "sanity" checks on user-provided data before using that data on sensitive operations. Many web vulnerabilities are described as Tainted Flow Attacks; examples include: SQL injection, cross-site scripting (XSS), malicious file inclusion, unwanted command executions, eval injections, unvalidated redirects, cross-channel scripting (XCS), and file system attacks [1, 2, 3, 4, 5]. To detect these kinds of attacks one needs to answer the following question: does the target program contain a path on which data flow from some input to a sensitive place without going through a sanitizer function? A sanitizer is a function that either "cleans" malicious data or warns about their potential threat. We call the previous question the *Tainted Flow Problem*. The Open Web Application Security Project [1] classifies three of these vulnerabilities, e.g., SQL injection, XSS, and unvalidated redirects, among the top ten security vulnerabilities found in web applications [5]. Furthermore, the annual SANS's report[2] estimates that SQL injection happened approximately 19 million times in July of 2009 only. Therefore, detection of potential vulnerabilities in web applications is an important problem.

The tainted flow problem was formalized by Ørbæk and Palsberg in 1997 as an instance of type-checking [6]. They proposed a type system to the typed λ-calculus, and proved that if a program type-checks, then it is free of tainted flow vulnerabilities. Ten years later, Jovanovic *et al.* provided an implementation of an algorithm that solves the tainted flow problem for PHP 4.0 in the Pixy tool. This algorithm was a dataflow version of Ørbæk and Palsberg's type system. It has an average $O(V^2)$, and a worst case $O(V^4)$ runtime complexity. Ørbæk and Palsberg's solution, when seen as a dataflow problem, admits a worst case $O(V^3)$ solution [6, p.30].

[1] `http://owasp.org`

[2] `http://www.sans.org/top-cyber-security-risks/origin.php`

This paper improves on the complexity of these previous results. The time and space complexity of the algorithm that we propose is, in the worst-case, quadratic on the number of variables in the source program. Key to achieve this lower complexity is the use of a program representation introduced by Bodik *et al.* [7] known as *Extended Static Single Assignment* (e-SSA). This representation can be computed in linear time on the program size and was originally proposed to eliminate array bound checks for Java programs. We describe the first attempt to use this representation to perform tainted flow analysis. As we will explain in Section 3.1, dataflow analyses usually store information associated with each variable at each program point, i.e., a point between consecutive instructions. An instruction might cause this information to change from one program point to the next; however, these changes usually are restricted to a single variable. The rest of the data associated with the other variables is redundant information. The e-SSA representation allows us reduce the amount of redundancy. Each variable is defined at most once, and the information associated with a variable is invariant along its entire live range; thus, we can bind this information irreversibly to the variable name. This paper brings forward the following contributions:

- An efficient algorithm to solve the tainted flow problem. A distinguishing feature of this algorithm is the use of the e-SSA representation to generate constraints. See Section 4.5.
- An implementation of the algorithm on top of `phc` [8, 9], an open source PHP compiler [3]. Our implementation of e-SSA is now part of the compiler's official distribution.
- An evaluation of the proposed approach on public PHP applications, including benchmarks used in previous works [10, 11, 3], and the consequent exposure of previously unknown vulnerabilities. See Section 5.

Our analysis can be generalized to other procedural languages. We have chosen PHP for two reasons. First, it is popular: PHP programs can be found in over 21 million Internet domains[4]. Second, it has been the focus of previous research on detection of vulnerabilities [10, 11], and benchmarks are easily available.

[3] `http://www.phpcompiler.org/`

[4] `http://php.net/usage.php`

2. Examples of Tainted Flow Attacks

A tainted flow attack is characterized by a subpath from a *source* to a *sink* function that does not include calls to *sanitizing functions*. A source function reads information from an input channel (e.g., from an HTML form) and passes it to the program. Sinks are functions that perform sensitive operations, such as writing information into the program's output channel (e.g., to a dynamically-generated webpage). Sanitizers are functions that protect the program. For instance, checking whether some untrusted information is safe, removing malicious contents from tainted data, or firing exceptions when necessary. The literature describes many kinds of tainted flow attacks; in this section, we explain two of these vulnerabilities in more detail: cross site scripting and SQL injection.

2.1. Cross-Site Scripting (XSS)

A cross-site scripting attack occurs when a user prints HTML content into a dynamically-generated page. An attacker uses this vulnerability to inject JavaScript code into the page, usually trying to steal cookie information to acquire session privileges. The program below illustrates this situation. In this case, the user provides the input "`<script>does.something.evil;</script>`" to the variable `name`:

```
<?php $name = $_GET['name']; echo $name; ?>
```

Note that a malicious JavaScript program could have been passed as argument instead of the text `does.something.evil`. Considering the command `echo` in PHP would evaluate its script argument, the negative consequence could be proportional to the script's knowledge and intent. A workaround for this threat is to strip HTML-related data from the user input. The function `htmlentities`, shown below, does the trick by replacing special characters with their HTML representations, e.g., this function replaces the symbol "<" by "`&lt;`". There exist public libraries that perform this type of sanitization (e.g., http://code.google.com/p/closure-templates/).

```
<?php $name = htmlentities($_GET['name']); echo $name; ?>
```

Cross-site scripting attacks fit into the tainted flow problem framework. A possible input configuration, in this case, would be:

Sources : `$_GET`, `$_POST`, . . .

Sinks : echo, print, printf

Sanitizers : htmlentities, htmlspecialchars, strip_tags

2.2. SQL Injection Attacks

SQL injection is another common type of tainted flow attack. In this case, an adversary uses the parameters of SQL queries to manipulate a database. The effect can go from reporting incorrect results to the user to modifying database contents. The program below contains a vulnerability of this kind.

```
<?php
  $userid = $_GET['userid'];
  $passwd = $_GET['passwd'];
  $result = mysql_query("SELECT userid FROM users WHERE
                 userid=$userid AND passwd='$passwd'");
?>
```

Note that this program does not sanitize its inputs. A malicious user could obtain access to the application by providing the text "1 OR 1 = 1 --" in the userid field. The double hyphen starts a comment in MySQL. The following query is obtained with the input variables replaced: SELECT userid FROM users WHERE userid=1 OR 1 = 1 -- AND passwd='ANY PASSWORD'. The execution of this query bypasses the authentication procedure, as the expression 1 = 1 is always true.

A workaround for this threat is to sanitize the variable userid to ensure that it only contains numerical characters; a task that we perform either casting it to integer or checking its value with functions like is_numeric. One can sanitize variable $passwd using the addslashes function, which inserts slashes (escape characters) before a predefined set of characters, including single quotes. A typical configuration of SQL injection is given below:

Sources : $_GET, $_POST, ...

Sinks : mysql_query, pg_query, *_query

Sanitizers : addslashes, mysql_real_escape_string, *_escape_string

3. Formal Definition and Previous Solution

Nano-PHP. We use the assembly-like Nano-PHP language to define the tainted flow problem. A label $l \in L$ refers to a program location and is associated to one instruction. A Nano-PHP program can be represented as a sequence of labels $l_1; l_2; \ldots; l_{exit}$. Figure 1 shows the six instructions of the language. We use the symbol $\otimes$ to denote any operation that uses a sequence of variables and is used in the context of a variable definition.

Semantics. We define the semantics of Nano-PHP programs with an abstract machine. The state M of this machine is characterized by a tuple (Σ, F, I), informally defined as follows:

Store Σ : $\mathit{Var} \to \mathit{Abs}$	e.g., $\{x_1 \mapsto \text{clean}, \ \ldots, \ x_n \mapsto \text{tainted}\}$
Code Heap F : $L \to [\mathit{Ins}]$	e.g., $\{l_1 \mapsto i_1 \ldots i_a, \ \ldots, \ l_n \mapsto i_b\}$
Instruction Sequence I : $[\mathit{Ins}]$	e.g., $i_5 i_6 \ldots i_n$

The symbol *Var* denotes the domain of program variables. The symbol *Abs* denotes the domain of abstract states $\{\perp, \text{clean}, \text{tainted}\}$. A variable that stores the abstract value $\perp$ is undefined, clean indicates the variable is clean (i.e., protected against tainted attacks), and tainted indicates that the variable may be used to create a tainted attack. The store Σ binds each variable name, say $x \in \mathit{Var}$, to an abstract value $v \in \mathit{Abs}$. The code heap F is a map from a program label to a sequence of instructions. Each sequence corresponds to one *basic block* from the Nano-PHP program. Only labels associated to entry basic block instructions appear in F. The list I denotes the next instructions for execution. We say that the abstract machine can *take a step* if from a state M it can make a transition to state M'. More

Name	**Instruction**	**Example from PHP**
Assignment from source	$x = \circ$	`$a = $_POST['content']`
Assignment to sink	$\bullet = v$	`echo($v)`
Simple assignment	$x = \otimes(x_1, \ldots, x_n)$	`$a = $t1 * $t2`
Branch	`bra` $l_1, \ldots, l_n$	general control flow
Filter	$x_1 =$ `filter`	`$a = htmlentities($t1)`

Validator	`validate` x, l_c, l_t	`if (!is_numeric($1))`
		`abort();`

Figure 1: The Nano-PHP syntax.

$$[\textsc{S-Source}] \qquad (\Sigma, F, x = \circ; S) \rightarrow (\Sigma\backslash[x \mapsto \text{tainted}], F, S)$$

$$[\textsc{S-Sink}] \qquad \frac{\Sigma \vdash v = \text{clean}}{(\Sigma, F, \bullet = v; S) \rightarrow (\Sigma, F, S)}$$

$$[\textsc{S-Simple}] \qquad \frac{\Sigma \vdash \sqcup(x_1, \ldots, x_n) = v}{(\Sigma, F, x = \otimes(x_1, \ldots, x_n); S) \rightarrow (\Sigma\backslash[x \mapsto v], F, S)}$$

$$[\textsc{S-Branch}] \qquad \frac{\{l_i\} \subseteq \text{dom}(F) \qquad F(l_i) = S' \qquad 1 \leq i \leq n}{(\Sigma, F, \texttt{bra}\ l_1, \ldots l_n; S) \rightarrow (\Sigma, F, S')}$$

$$[\textsc{S-Filter}] \qquad (\Sigma, F, x = \texttt{filter}; S) \rightarrow (\Sigma\backslash[x \mapsto \text{clean}], F, S)$$

$$[\textsc{S-ValidC}] \qquad \frac{\Sigma \vdash x = \text{clean} \qquad \{l_c\} \subseteq \text{dom}(F) \qquad F(l_c) = S'}{(\Sigma, F, \texttt{validate}(x, l_c, l_t); S) \rightarrow (\Sigma, F, S')}$$

$$[\textsc{S-ValidT}] \qquad \frac{\Sigma \vdash x = \text{tainted} \qquad \{l_t\} \subseteq \text{dom}(F) \qquad F(l_t) = S'}{(\Sigma, F, \texttt{validate}(x, l_c, l_t); S) \rightarrow (\Sigma, F, S')}$$

Figure 2: Operational semantics of Nano-PHP.

formally, we write $M \rightarrow M'$. We say that the machine is *stuck* at M if it cannot make any transition from M.

Figure 2 illustrates the transition rules describing the semantics of Nano-PHP programs. Rule S-Source states that an assignment from source binds the left-hand side variable to the tainted abstract state. Rule S-Sink is the only one that can cause the machine to get stuck: the variable on the right hand side *must* be bound to clean in order to execute a safe assignment to sink. Rule S-Simple says that, the new store of variable x after an assignment of the form $x = \otimes(x_1, x_2, \ldots, x_n)$ is obtained by folding the meet operation (as described in Table 1) across the elements of the list of variables in the right-hand side of the expression, i.e.: $x_1 \wedge x_2 \ldots \wedge x_n$. This rule conservatively propagates the tainted information to the new definition of x: if at least one of the variables used, e.g., $x_i, 1 \leq i \leq n$, is tainted so will be the new store of x. Rule S-Branch defines a non-deterministic branch choice:

$\wedge$	$\perp$	clean	tainted
$\perp$	$\perp$	clean	tainted
clean	clean	clean	tainted
tainted	tainted	tainted	tainted

Table 1: Meet operator over pairs of abstract values.

the machine chooses one target in a range of possible labels and branches execution to the instruction at this label.

Nano-PHP organizes the sanitizer function in two groups: filters and validators. Filters correspond to functions that take a value, typically of string type, and return another value without malicious fragments from the input. For simplicity we do not show the input parameter in the syntax of Nano-PHP. Rule S-Filter shows that an assignment from a filter binds the variable on the left side to the clean state. We can use this syntax to define assignments from constants (e.g., $v = 1$). Validators are instructions that combine branching with a boolean function that checks the state for tainting. The instruction $\texttt{validate}(x, l_c, l_t)$ has two possible outcomes. If x is bound to the clean state, the machine branches execution to $F(l_c)$. If x is bound to the tainted state, execution branches to $F(l_t)$. Again, we omit the boolean function itself from the syntax for simplicity. Rules S-ValidC and S-ValidT define these cases. We assume that in any Nano-PHP program every variable must be defined before being used; therefore, we rule out the possibility of passing x to a validator when $\Sigma \vdash x = \perp$.

Important consideration about functions. In this paper we describe an intraprocedural analysis. Thus, we conservatively consider that input parameters and the return values of called functions are all definitions from source. A context insensitive, interprocedural version of the proposed algorithms can be produced by creating assignments from actual to formal parameters.

The problem. We define the tainted flow problem as follows.

Definition The Tainted Flow Problem

Instance: a Nano-PHP program P.
Problem: determine if the machine can get stuck while executing P.

3.1. Data Flow Analysis.

Given a Nano-PHP program, we can solve the tainted flow problem using a *forward-must* data flow analysis. We define a lattice $(Abs, <)$ by augmenting the set Abs with the ordering $\bot <$ clean $<$ tainted. Table 1 shows the meet operator $\wedge$ over this lattice. The map lattice $(Var \to Abs, <')$ is obtained with the typical lifting of the lattice associated to Abs. Recall that the set Var is finite. We represent dataflow information with the function $[\![\_]\!] : L \to L \to Var \to Abs$. This function associates to each program point (l, l') a map storing the abstract values of variables. We use the notation $[\![l_1, l_2]\!]$ to denote information at (l_1, l_2). It abbreviates the function application $([\![\_]\!]l_1)l_2$. We prefer to associate our transfer functions to pairs of labels, instead of to the In and Out sets of more classic approaches, to simplify the description of a path sensitive analysis. In this way we can bind information directly to the edges of the control flow graph, and an instruction such as `validate` can generate different information at different edges.

Table 2 defines the transfer functions $(Var \to Abs) \to (Var \to Abs)$ associated to each instruction. The initial state of the analysis associates undefined to all program variables at every point, i.e., $[\![\_]\!] = \lambda l_1 \,.\, \lambda l_2 \,.\, \lambda v \,.\, \bot$. We let $PRED(l)$ be the set of labels l_i immediately before label l, and define the auxiliary function $Meet$ as follows:

$$Meet(l) = \bigwedge [\![l_i, l]\!] \;, \qquad l_i \in PRED(l)$$

Given two functions $[\![k', k]\!]$ and $[\![l', l]\!]$, we define $\bigwedge\ \{[\![k', k]\!], [\![l', l]\!]\}$ as $\lambda v \;.([\![k', k]\!]v) \wedge ([\![l', l]\!]v)$, with $\wedge$ given by Table 1. The combined transfer function $tr : [\![\_]\!] \to [\![\_]\!]$ is defined as usual with the composition of all individual transfer functions. Function tr admits fix-points as the lattice is finite and all individual transfer functions are monotone [14, Ch.9].

The meet operation denotes accumulation of information across control flow edges. In this case, information flows from the predecessor edges of a node. Note that we define operation $Meet$ over a map lattice. Informally, the semantics of this operation is to apply $\wedge$ over elements on the image of the functions according to the definition in Table 1. For example $\{x \mapsto \text{clean}, y \mapsto \text{clean}\} \wedge \{x \mapsto \text{tainted}, y \mapsto \bot\} = \{x \mapsto \text{clean} \wedge \text{tainted}, y \mapsto \text{clean} \wedge \bot\}$. Notice that this dataflow system is path-sensitive, given the way that we handle `validate` x, l_c, l_t. Information flows differently to each successor label, l_c or l_t, depending on the parameter x been clean or tainted.

	instruction	$[\![\_]\!]$
1	$x = \circ$	$[\![l, l_+]\!] = \mathit{Meet}(l) \setminus [x \mapsto \text{tainted}]$
2	$\bullet = x$	$[\![l, l_+]\!] = \mathit{Meet}(l)$
3	$x = \otimes(x_1, \ldots, x_n)$	$[\![l, l_+]\!] = \mathit{Meet}(l) \setminus [x \mapsto \mathit{Meet}(l)(x_1) \wedge \ldots \wedge \mathit{Meet}(l)(x_n)]$
4	$\texttt{bra}\ l_1, \ldots, l_n$	$\forall i, 1 \leq i \leq n, [\![l, l_i]\!] = \mathit{Meet}(l)$
5	$x = \texttt{filter}$	$[\![l, l_+]\!] = \mathit{Meet}(l) \setminus [x \mapsto \text{clean}]$
6	$\texttt{validate}\ x, l_c, l_t$	$[\![l, l_c]\!] = \mathit{Meet}(l) \setminus [x \mapsto \text{clean}], \quad [\![l, l_t]\!] = \mathit{Meet}(l)$

Table 2: dataflow equations to solve the Tainted Flow Problem.

If $\mathit{USE}(\mathrm{x})$ is the set of instructions where variable x is used, then we say that an instruction i *influences* a variable v if either i defines v, or i defines x, and some instruction $i' \in \mathit{USE}(x)$ influences v. The dataflow equations provide the invariant stated in Theorem 3.1. We flag a security vulnerability if a Nano-PHP program contains an instruction $l : \bullet = v$, and $[\![l, l_+]\!](v) = \text{tainted}$.

Theorem 3.1. *If $[\![l_1, l_2]\!](v) = \mathit{tainted}$, then v is created by an instruction influenced by an assignment from source.*

Figure 3 shows the result of a dataflow analysis. We let `DB` to denote a global database, and we assume that `DB.get` might produce tainted data. The function `DB.isMember` works as a validator. We have replaced a call to `DB.hasParent` by the simple branch at l_6, as this operation does not create new data. Similarly, we have replaced the call to `DB.getParent` by $v = \otimes(v)$. Because our instruction set does not contain a halt, we use l_8, a label jumping to itself, to mark the end of the program. We show the maps produced by the dataflow analysis on the edges of the Nano-PHP program. In this program the dataflow analysis obtains a fix-point in two iterations. The example contains a tainted flow vulnerability, given by the path $l_4 \to l_5 \to l_6 \to l_3$. At l_4 we read variable x, e.g., `$x = $_POST['$v']`, and at l_3 we feed it to a sink function, e.g., `echo($x)`. Note that variable v cannot be used in a tainted flow attack, because it is sanitized by the function `DB.isMember`.

```
<?php
$v = DB.get($_GET['child']);
$x = "";
if (DB.isMember($v)) {
  while (DB.hasParent($v)) {
    echo($x);
    $x = $_POST['$v'];
    $v = DB.getParent($v);
  }
  echo($v);
}
?>
```

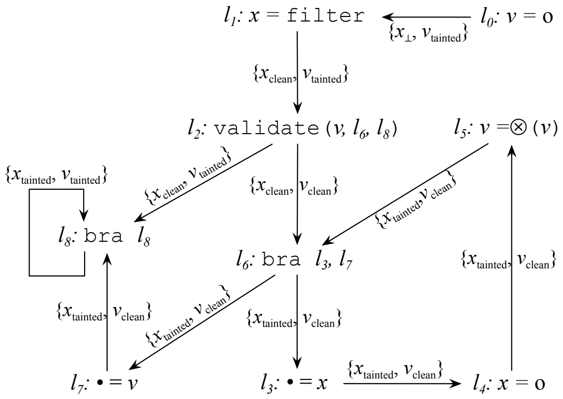

l_1: x = filter
$\{x_\perp, v_{\text{tainted}}\}$
l_0: v = o
$\{x_{\text{clean}}, v_{\text{tainted}}\}$
l_2: validate(v, l_6, l_8)
l_5: v =⊗(v)
$\{x_{\text{tainted}}, v_{\text{tainted}}\}$
$\{x_{\text{clean}}, v_{\text{tainted}}\}$
$\{x_{\text{clean}}, v_{\text{clean}}\}$
$\{x_{\text{tainted}}, v_{\text{clean}}\}$
l_8: bra l_8
l_6: bra l_3, l_7
$\{x_{\text{tainted}}, v_{\text{clean}}\}$
$\{x_{\text{tainted}}, v_{\text{clean}}\}$
$\{x_{\text{tainted}}, v_{\text{clean}}\}$
$\{x_{\text{tainted}}, v_{\text{clean}}\}$
$\{x_{\text{tainted}}, v_{\text{clean}}\}$
l_7: • = v
l_3: • = x
$\{x_{\text{tainted}}, v_{\text{clean}}\}$
l_4: x = o

Figure 3: A simple PHP program (left), and its equivalent Nano-PHP version (right), augmented with the result of dataflow analysis.

Complexity.. We can solve this dataflow analysis using the chaotic iteration model [15]. If the CFG of the input program has I instructions and V variables then we can perform $O(I \times V)$ iterations. Each union is $O(V)$, and we may have $O(I)$ unions per iteration. Thus, our dataflow analysis has complexity $O(V^2 \times I^2)$. However, it is possible to speedup the algorithm executing the transfer functions in a topological order of the program's dominator tree [15]. In particular, Palsberg [16] gives an $O(V^3)$ type-inference algorithm that solves the tainted flow problem. In practice, this dataflow analysis is $O(V \times I)$ [15, p.209].

4. The Proposed Solution

This section describes our proposed solution to the tainted flow problem. Our approach is organized in three parts as described below.

1. Convert the input program to the *Extended Static Single Assignment* (e-SSA) form. The construction of the dominator tree is $O(V\alpha(V))$, where V is the number of variables in the source program and α is the inverse Ackermann function, normally regarded as constant, and the insertion of ϕ-functions is $O(V^2)$, yet linear in practice [15, p.408].
2. Traverse the e-SSA-form program collecting use-chains: $O(V)$.

3. Use the algorithm in Figure 8 to find tainted flow vulnerabilities: $O(V^2)$, but $O(V)$ in practice.

4.1. E-SSA form is key to Fast Path-Sensitive Tainted Flow Analysis

We use the *Extended Static Single Assignment* (e-SSA) representation to simplify our tainted flow analysis. The e-SSA program representation is a superset of the well known *Static Single Assignment* (SSA) form [17]. This representation has been used by Bodik *et al.* [7] to eliminate array bound checks. Its main advantage, in our case, is the possibility of acquiring useful information from the outcome of conditional tests, and then binding this information directly to variables, instead of pairs of variables and program points. We convert a Nano-PHP program to e-SSA form using the algorithm below:

1. For each instruction $i =$ `validate` x, l_c, l_t:
   (a) replace i by a new instruction `validate` x, x_c, l_c, x_t, l_t, where x_c and x_t are fresh variables;
   (b) rename every use of x dominated by l_c to x_c. A label l dominates a use of variable x at label l_u if, and only if, every path from the program's entry point to l_u goes across l.
   (c) rename every use of x dominated by l_t to x_t;
2. Convert the resulting program into SSA form. For a fast algorithm, see Appel and Palsberg [15, p.410].

In order to represent Nano-PHP program in e-SSA form, we modify the syntax of this language in two ways. First, we add ϕ-functions to the language. These special instructions are an abstraction first introduced by Cytron *et al.* [17] to represent SSA-form programs. ϕ-functions are used at control-flow join points, and they receive as parameter one variable name associated to each control-flow predecessor. A ϕ-function such as $x_n = (x_1, \ldots, x_m)$, placed at label l has the effect of assigning $x_i, 1 \leq i \leq m$ to x_n, depending on which predecessor of l was last visited before execution reaches l. The use of a variable in SSA-form programs is associated to only one definition. Thus, to convert a program into the SSA form, we rename each definition of a variable v to a different name, and join definitions of v that are alive at a common program point by ϕ-functions. These new ϕ-functions produce fresh definitions of v; thus, the process continues until the program stabilizes. There exist almost linear time algorithms to convert programs to SSA-form [18]. E-SSA-form programs are also SSA-form programs; thus, they have the property that each variable has only one definition.

E-SSA-form programs are also SSA-form programs: they also have the property that each variable has only one definition. The difference is that while SSA *merges* live-ranges of many related variables at the point of a merge of conditional branches, e-SSA, in addition, *splits* the live-ranges of one variable at the point of a conditional branch. In result, the e-SSA representation provide path-sensitiveness as it allows one to obtain static information more precisely from the outcome of conditionals. Operationally, we modified the syntax and semantics of the validator instruction. The syntax becomes `validator` (x, x_c, l_c, x_t, l_t) [5]; the semantics we show below.

$$[\text{S-E{\scriptsize SSA}C}] \quad \frac{\Sigma \vdash x = \text{clean} \qquad \{l_c\} \subseteq \text{dom}(F) \qquad F(l_c) = S'}{(\Sigma, F, \texttt{validate}(x, x_c, l_c, x_t, l_t); S) \to (\Sigma \setminus [x_c \mapsto \text{clean}], F, S')}$$

$$[\text{S-E{\scriptsize SSA}T}] \quad \frac{\Sigma \vdash x = \text{tainted} \qquad \{l_t\} \subseteq \text{dom}(F) \qquad F(l_t) = S'}{(\Sigma, F, \texttt{validate}(x, x_c, l_c, x_t, l_t); S) \to (\Sigma \setminus [x_t \mapsto \text{tainted}], F, S')}$$

The validator splits the live range of variable x in two parts, depending on whether or not its abstract value is tainted. Note that when converting a program into e-SSA form, we rename every use of x in labels dominated by l_c to x_c, and rename every use of x in labels dominated by l_t to x_t. The new instruction has the following semantics: Rule S-EssaC says that a validator, upon receiving a clean variable x, guarantees that the variable will be clean henceforth. Given that every use of x dominated by l_c has been renamed to x_c beforehand, we simply continue the program execution in an environment where x_c is bound to clean. Rule S-EssaT does the opposite: if a validator fails on a variable x, we know that x is tainted; hence, we continue the program execution in an environment where x_t is bound to tainted.

Figure 4 illustrates that e-SSA enables one to associate unique constraints to variables, as opposed to program points. The original program in Figure 3 contains two variables, x and v. We know that these variables are clean in some program points, but not in all. The e-SSA representation allows us

[5] Bodik *et al.* use special instructions called π-functions to create x_c and x_t [7]

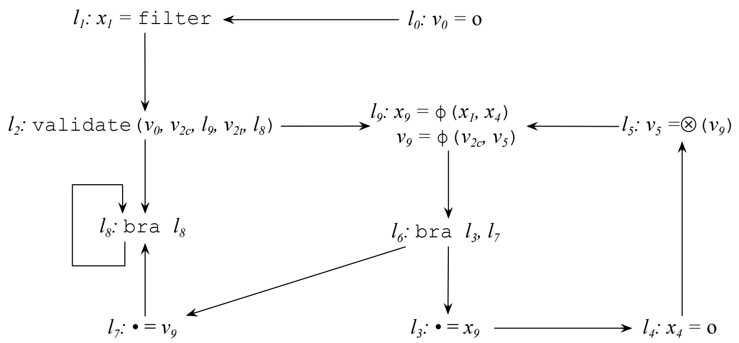

l_1: x_1 = filter
l_0: v_0 = o
l_2: validate (v_0, v_{2c}, l_9, v_{2t}, l_8)
l_9: $x_9 = \phi(x_1, x_4)$
$v_9 = \phi(v_{2c}, v_5)$
l_5: $v_5 = \otimes(v_9)$
l_8: bra l_8
l_6: bra l_3, l_7
l_7: $\bullet = v_9$
l_3: $\bullet = x_9$
l_4: x_4 = o

Figure 4: The example from Figure 3 converted into e-SSA form. Note the difference in `validate` and that information is *not* associated to the edges of the graph.

to identify these program points precisely. The modified program has five variables created after v: $\{v_0, v_5, v_9, v_{2c}, v_{2t}\}$, plus three variables created after x: $\{x_1, x_4, x_9\}$. Let's consider the first group of variables. Given that v_0 is produced by source assignment, we know that it is tainted. Variable v_{2c} must be necessarily clean, as it is produced by the validation of v_9. On the other hand, v_{2t} must be necessarily tainted, for the opposite reason. Variable v_5, which results from the application of an operation – assignment – on a clean variable, is also clean. Finally, v_9, which may be assigned either a clean or a tainted value, is tainted, as this is the most conservative choice to detect security vulnerabilities.

4.2. Tainted Analysis as Graph Reachability

Given a Nano-PHP program P, we represent it as a graph G, in which each node $n_v \in G$ denotes a variable $v \in P$. We build the reachability graph directly from the e-SSA-form Nano-PHP program. Each particular type of instruction produces a specific configuration of nodes in the reachability graph, as Table 3 shows. Roughly, there is an edge linking n_u to n_v if information flows from variable u to v. Notice that, were it not for filters and validators, our reachability graph would represent the def-use chains of the Nano-PHP program. The program from Figure 4 gives origin to the reachability graph in Figure 5.

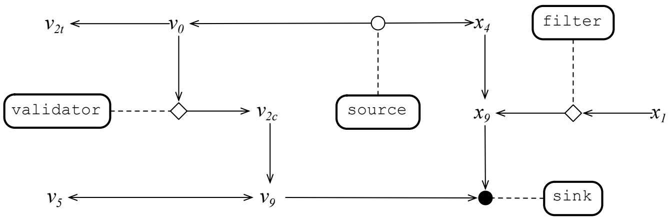


Figure 5: The reachability graph built after the program in Figure 4.

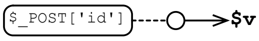
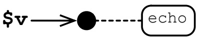
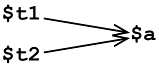
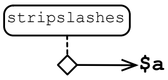
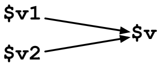
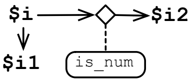
Instruction	Example	Nodes
$v = \circ$	`$v = $_POST['id']`	
$\bullet = v$	`echo($v)`	
$v = \otimes(v_1, \ldots, v_2)$	`$a = $t1 * $t2`	
$v =$ `filter`	`$a = stripslashes($t)`	
$v = \phi(v_1, \ldots, v_2)$	`$v = phi($v1, $v2)`	
`validate` (v, v_c, l_c, v_t, l_t)	`if(is_num($i))`	

Table 3: Mapping program instructions to nodes in the reachability graph.

Below we rephrase the tainted flow problem as an instance of graph reachability. The traversal of the reachability graph is related to the notion of program slicing [19]. Any node u that reaches a node v is part of the program slice that defines the behavior of v.

Definition The Tainted Flow Problem as Graph Reachability

Instance: a graph G that describes a Nano-PHP program P.

Problem: determine if G contains a path from a source to a sink that does not cross any sanitizer.

4.3. Handling Arrays and Class Fields

Our analysis is field sensitive. The `phc` compiler gives us enough infrastructure to distinguish assignments to different fields inside an object. Thus, we can handle fields in the same way that we handle ordinary variables. The treatment of arrays is more complicated. In PHP, arrays can be indexed with either integers or strings, can grow and shrink dynamically and may not be contiguous. Array cells can also store composite types such as other arrays [6].

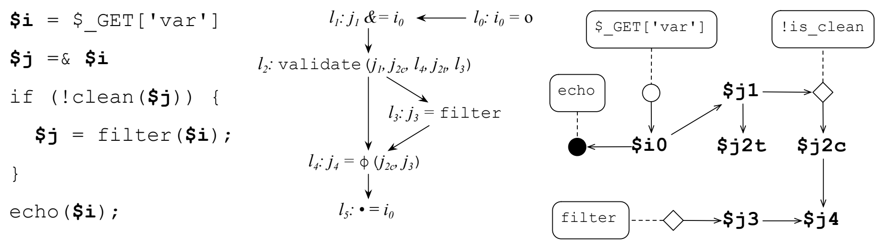
The `phc` compiler provides a def-use chain for array elements. The compiler tracks assignments to array indexes the same way as done for scalar assignments. If an assignment uses a constant statically known to index an array, e.g., `_POST["name"] = 1`, then we can treat the pair (`_POST`, `"name"`) as a single variable. We consider the array itself, e.g., `_POST` as clean, since it cannot be directly used in an attack. Nevertheless, there are cases when we are unable to statically determine the value used to index an array, e.g., `_POST[N] = 1`.

When reading an array element with an unknown index, we check whenever we already inferred any element of this array as tainted. In this case, we conservatively consider this array access as tainted in order to avoid missing true positives, otherwise we consider it clean. When writing to an unknown position of an array we check whenever we are assigning a tainted value to it. If that is the case, then we conservatively mark every element of this array as tainted. Otherwise we ignore the assignment and the analysis continues. Although this strategy may increase the number of false-positives, it is better than treating the entire memory heap as a single unit.

4.4. Addressing Aliasing with HSSA

Aliasing is a phenomenon typical of imperative languages, in which two names reference the same memory location. Aliasing complicates static analyses because it requires the analyzer to understand that updates in the state of a variable may also apply to other variables. To see the implications of aliasing on tainted flow analysis, let's consider the PHP program in Figure 6

[6] `http://www.php.net/manual/en/language.types.array.php`



```
$i = $_GET['var']
$j =& $i
if (!clean($j)) {
  $j = filter($i);
}
echo($i);
```

Figure 6: An example of how aliasing complicates the tainted flow analysis. In the right side we show the reachability graph built for the e-SSA form program.

(Left). Assuming that `$_GET` is a source and `echo` is a sink, then the program is logically bug free. That is, the name `$i`, which is used in a sink, has been sanitized as name `$j`, because both names, `$i` and `$j` represent the same variable. The ordinary e-SSA representation will not catch this subtlety, as Figure 6 shows. There is a clear path from `$i0` to the sink that does not go across any sanitizer.

In order to deal with aliasing we use an augmented flavor of the e-SSA representation, that we derive from a representation called *Hashed Static Single Assignment (HSSA)* form [20]. This last program representation is used internally by `phc` [8, Sec 6.5], our baseline compiler. For each assignment $v = E$ in a SSA-form program, the equivalent HSSA-form program contains an assignment $(v, a_1, \ldots, a_n) = E$, where $a_1, \ldots a_n$ are the aliases of v at the assignment location. Following this strategy, our augmented representation generates new names for each variable created by a sanitizer. The literature contains a plethora of methods to conservatively estimate the set of aliases of a variable. We use the flow sensitive, interprocedural analysis [21] that we obtain from `phc`. Moving on with our example, Figure 7 shows the program and the reachability graph after augmenting the e-SSA form program in Figure 6 with the results of alias analysis. In the new reachability graph there is no path from a source to a sink that does not go across a sanitizer. Thus, we report that the program is bug-free.

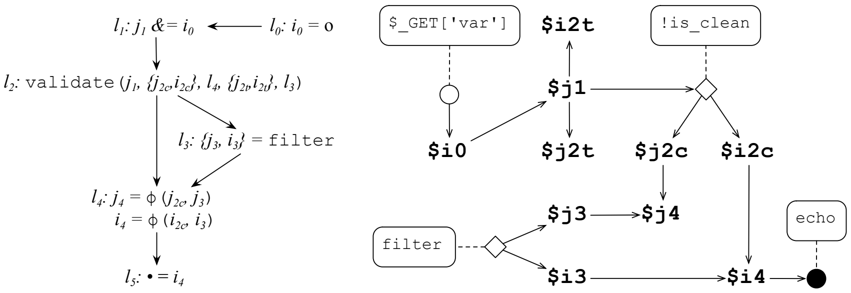

l_1: j_1 &= i_0
l_0: i_0 = o
l_2: validate (j_1, {j_{2c},i_{2c}}, l_4, {j_{2t},i_{2t}}, l_3)
l_3: {j_3, i_3} = filter
l_4: j_4 = ϕ (j_{2c}, j_3)
i_4 = ϕ (i_{2c}, i_3)
l_5: • = i_4
$_GET['var']
$i2t
!is_clean
$j1
$i0
$j2t
$j2c
$i2c
$j3
$j4
echo
filter
$i3
$i4

Figure 7: (Left) input program in e-SSA form augmented with the results of alias analyses. (Right) final reachability graph.

4.5. A Solution Quadratic in Time and Space

The function *markTaintedVars* in Figure 8 finds security bugs in e-SSA-form Nano-PHP programs. We use SML/NJ's syntax plus Erlang-style guards in pattern matching, as in *taintUse*. *MarkTaintedVars* simulates a traversal of the reachability graph from Section 4.2, but it does not build the graph. Instead, it traverses *use-chains*. We let *USE*(x) be defined as in Section 3.1. The use-chains of a variable v are the paths created by successive applications of the *USE* function. For instance, in Figure 6 (Right), the use-chains of variable `j1` are the paths `j1`→`j2t`, and `j1`→`j2c`→`j4`.

Function *markTaintedVars* has three parameters: a set $\{i, i_1, \ldots, i_n\}$ of instructions to process, an environment Σ that maps variables to clean or tainted, and a set of visited instructions, which we keep to avoid visiting the same instruction twice. When properly initialized, *MarkTaintedVars* returns a new environment Σ' that maps to tainted every variable v in any path from a source instruction to a sink instruction. Theorem 4.1 formalizes this property. We say that a variable v *reaches* an instruction i if either $i \in \mathit{USE}(v)$, or some $i' \in \mathit{USE}(v)$ defines a variable v' that reaches i. We define the notion of *influence* as in Section 3.1. We initialize *markTaintedVars* with the instructions that assign variables from source, an environment Σ that maps every variable to clean, plus an empty set of visited instructions. For instance, if we apply *markTaintedVars* onto the program in Figure 4, then we start it with the set $I_\circ = \{v_0 = \circ, x_4 = \circ\}$.

Theorem 4.1. *Let $I_\circ$ be the set of assignments from source in a Nano-PHP program. If $\Sigma' = \mathit{markTaintedVars}\ I_\circ\ (\lambda v.\mathit{clean})\ \emptyset$, then $\Sigma'[v] = \mathit{tainted}$ if,*

```
fun $taintUse$ $\_$ $\{\ldots,(\bullet = x),\ldots\}$ $\Rightarrow$ true
 | $taintUse$ $\Sigma$ $\{\ldots,(x = \otimes(\ldots)),\ldots\}$ $\wedge$ $\Sigma \vdash x = \text{tainted}$ $\Rightarrow$ true
 | $taintUse$ $\Sigma$ $\{\ldots,(x = \phi(\ldots)),\ldots\}$ $\wedge$ $\Sigma \vdash x = \text{tainted}$ $\Rightarrow$ true
 | $taintUse$ $\Sigma$ $\{\ldots,(\texttt{validate}(\_,\_,\_,x,\_)),\ldots\}$ $\wedge$ $\Sigma \vdash x = \text{tainted}$ $\Rightarrow$ true
 | $taintUse$ $\_$ $\_$ $\Rightarrow$ false
fun $markTaintedVars$ $\emptyset$ $\Sigma$ $\_$ $\Rightarrow$ $\Sigma$
 | $markTaintedVars$ $\{i, i_1, \ldots, i_n\}$ $\Sigma$ $V$ $\Rightarrow$
    let
      val $V' = \{i\} \cup V$
      fun $doUseChainSearch$ $v$ =
        let
          val $N = USE(v) \setminus V'$
          val $\Sigma' = markTaintedVars$ $(\{i_1, \ldots, i_n\} \cup N)$ $\Sigma$ $V'$
        in
          if $taintUse$ $\Sigma'$ $USE(v)$
          then $\Sigma'[v \mapsto \text{tainted}]$
          else $\Sigma'$
        end
    in
      case $i$ of
        $\bullet = x \rightarrow markTaintedVars$ $\{i_1, \ldots, i_n\}$ $\Sigma[x \mapsto \text{tainted}]$ $V'$
        $x = \circ \rightarrow doUseChainSearch$ $x$
        $x = \otimes(\ldots) \rightarrow doUseChainSearch$ $x$
        $x = \phi(\ldots) \rightarrow doUseChainSearch$ $x$
        validate $x, x_c, l_c, x_t, l_t \rightarrow doUseChainSearch$ $x_t$
    end
```

Figure 8: The algorithm that finds security bugs in Nano-PHP programs.

and only if, v is influenced by an assignment from source, and v reaches an assignment to sink.

MarkTaintedVars processes each instruction forwardly, i.e., a variable x is tainted if any of the instructions that uses it defines a tainted variable. We use the auxiliary function *taintUse* to check if any of the instructions in the forward slice of a variable x defines a variable that has been set to tainted in the environment. We stop traversing use-chains at sink instructions. Neither *markTaintedVars* nor *taintUse* deals with switches or filter instructions.

These instructions do not define nor use tainted variables. *MarkTaintedVars* and the dataflow equations from Figure 2 are not equivalent. The dataflow equations taint any variable that is influenced by an assignment from source, whereas *MarkTaintedVars* only taints those that also reach an assignment to sink. However, from Theorems 3.1 and 4.1 we see that if a Nano-PHP program allows a source function to influence a sink operation, then both algorithms will find it.

Complexity. The function *markTaintedVars* is quadratic in time and space. Because *markTaintedVars* keeps the use-chains of every variable, this function uses $O(V \times I)$ space, where V is the number of variables in the input program, and I is the number of instructions in this program. The function is recursively called at most once per each program instruction. When the function is called, it might do a linear search on the use-chain of a variable, inside the function *doUseChainSearch*. Therefore, this function has time complexity $O(I^2)$.

5. Experiments

We have implemented the dataflow analysis discussed in Section 3 and our e-SSA based analysis from Section 4 on top of the `phc` open source compiler [8, 9], which is written in C++. In this section we compare the runtime of both implementations. The `phc` compiler, started in 2005 by Edsko de Vries and John Gilbert, currently uses our implementation of e-SSA as an internal representation. Our implementation of dataflow analysis, which we have based on the version available in the Pixy tool, uses a standard working list algorithm, and runs on the CFG of the input program sorted topologically. This implementation pattern is standard in compiler design [15, pag.360].

Benchmarks: We have run our analysis on 20,900 files publicly available in 30 PHP content management systems (CMS). Most of these applications have been used in previous works [3, 10, 11]. The names of these applications are given in Figure 9. In this section we show results for 13,297 files out of the 20,900 inputs (63.6%). The omissions are due to the fact that `phc`, being a static compiler, is not able to analyze some features of PHP, such as dynamic file inclusion or dynamic code evaluation. None of these failures are due to our implementations, i.e., they occur independently of our analysis. A detailed account of each `phc` failure is provided by Rimsa [13].

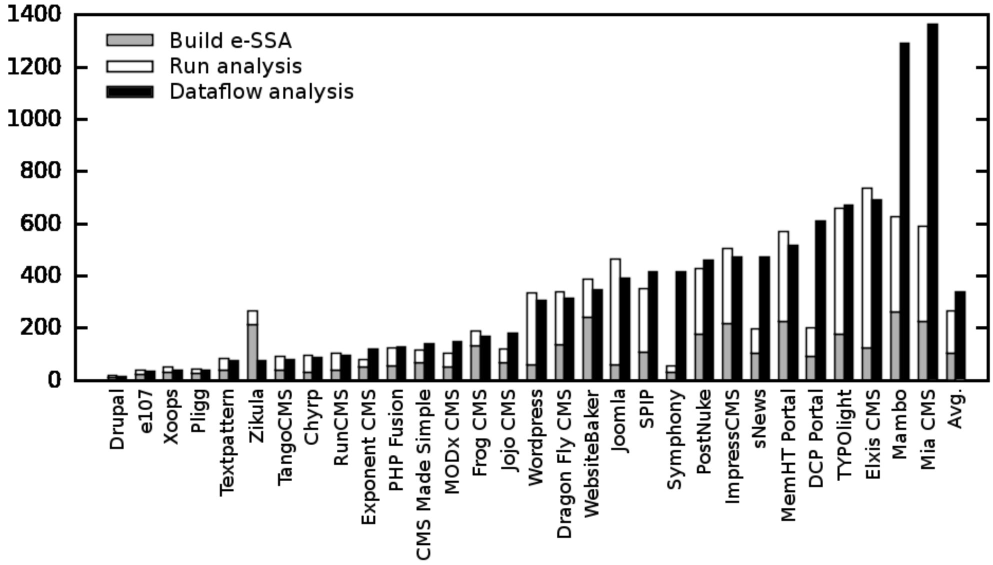


Figure 9: Average execution time (ms) per benchmark for dataflow and e-SSA-based analyses. Bars are sorted by the time to run the dataflow based analysis.

Set up: Currently our tool reads a configuration file that determines which functions (user defined or from libraries) are sinks, sources and sanitizers. For these experiments we use a configuration file that identifies cross-site scripting attacks, which we describe in Section 2.1. Notice that by properly pointing sources, sinks and sanitizers our analysis can be easily modified to handle other vulnerabilities, such as SQL injections (Section 2.2).

Efficiency: We compare the time to run the dataflow analysis (Section 3) and the time to run our sparse analysis (Section 4). We run the dataflow analysis on the original program, before the conversion to SSA (and e-SSA) form. In order to produce e-SSA form programs, we start from a non-SSA form program, and augment it with special instructions, i.e., σ and ϕ-functions [7, 17]. Figure 9 shows that the e-SSA based approach is faster than the dataflow approach as the size of the input functions grow. Each bar is the average sum of the times to process each function of the benchmark, over 10 runs. On the average, our sparse analysis is 28% faster than the traditional dataflow approach. We measure the time to analyze each func-

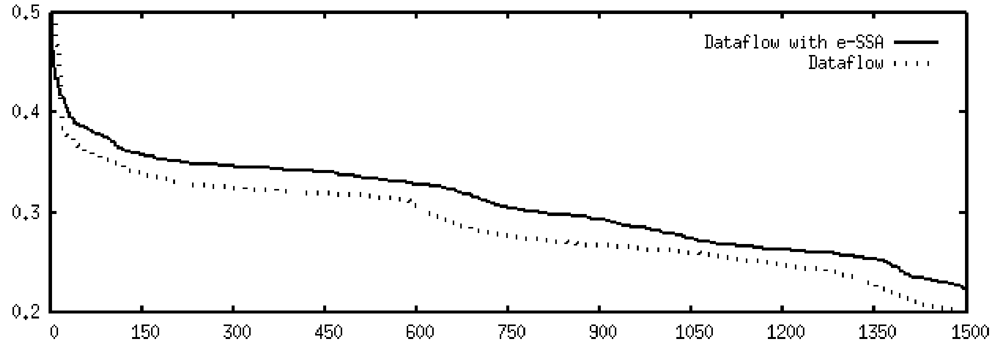


Figure 10: The 1,500 functions in which our e-SSA based algorithm run in the fastest time. We have ordered the functions according to our analysis runtime. The dataflow approach was 7.3% faster in this scenario.

tion individually, and we do not consider functions containing less than 100 assembly instructions, for in this case time measurements are too imprecise. Our benchmarks have provided us with 1,622 functions above this threshold. The largest function that we have analyzed contains 1,141 instructions.

In order to see that the sparse approach scales better than the dense analysis we have checked the runtime of both strategies in small and large functions. Figure 10 shows the run time of our analysis for the 1,500 functions that had the fastest processing time. These are very small functions, ranging from 1 to 43 basic-blocks. As Figure 9 shows, the dataflow analysis is faster in this scenario, because our approach has to bear the cost of initializing the data-structure that holds the use-chains. On the average, for these small functions the dataflow approach was 7.3% faster.

We chose 1,500 functions to build Figure 10 because, as we move towards the left corner of the chart our analysis starts delivering faster times. This tendency accentuates, as the size of the functions grows. Figure 11 shows the processing time of the 50 functions that our algorithm took the longest time to process. Our sparse approach was 20.3% faster in this scenario. The largest function that we have analyzed contains 378 basic blocks a small size to stress the runtime of both solutions. We speculate that once we cross the boundaries of functions, and start analyzing whole PHP applications, which might contain thousands of functions, and millions of lines of code, our

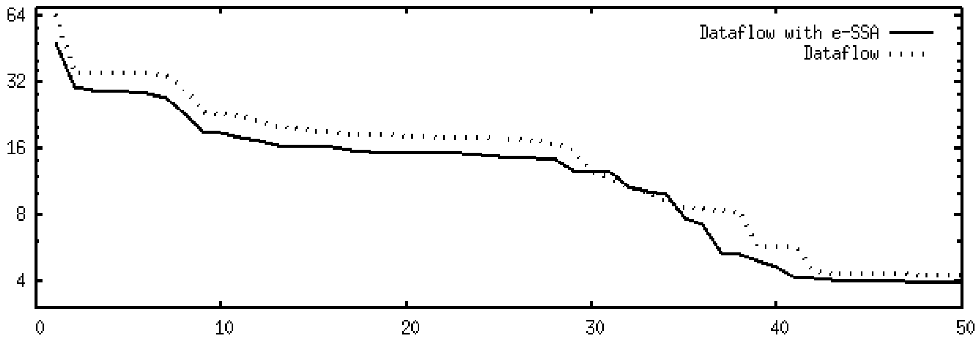


Figure 11: The 50 functions in which our e-SSA based analysis had the slowest runtimes. Functions are ordered by processing time. Our algorithm was 20.3% faster than the dataflow approach.

benchmark	version	files					warnings	
		total		processed		affected	TP	FP
		F	LOC / F	F	LOC / F			
MODx	1.0.3	472	231	308	228	3	1	1
Exponent	0.97	3456	42	2833	32	3	28	11
DCP Portal	7.0 beta	535	97	392	61	7	5	11
Pligg	1.0.4	380	146	179	154	3	1	0
RunCMS	2.1	737	134	361	86	2	1	6
avg.	-	-	-	-	-	3.60	7.20	5.8

Table 4: Precision results. F is the number of files, and LOC/F is the number of lines of PHP code per file. Affected is the number of files containing tainted flow vulnerabilities. TP are true positives, and FP are false positives.

analysis will be significantly more efficient than the dataflow approach. Our base compiler, `phc`, does not give us enough infra-structure to perform whole program analyses, but its community is working to overcome this limitation.

Precision: Both our e-SSA based analysis and the dataflow analysis have succeeded on the same inputs, reporting 63 warning messages across 25 distinct PHP files. Table 4 details these numbers for the subjects that contain confirmed vulnerabilities. Manual inspection of each of these warnings revealed actual vulnerabilities in 36 of these reports, i.e., a 45% false positive ratio. The false positives are due to the lack of whole program analysis, which force us to assume that every function parameter is tainted. We used this list of bugs to perform cross-site scripting attacks in 9 distinct PHP files. To the best of our knowledge, none of these vulnerabilities have been previously reported. We have submitted all these vulnerabilities to the bugtraq at `http://www.securityfocus.com/`. For a detailed account of each bug, see Rimsa [13, 22].

5.1. An example of a real-world bug

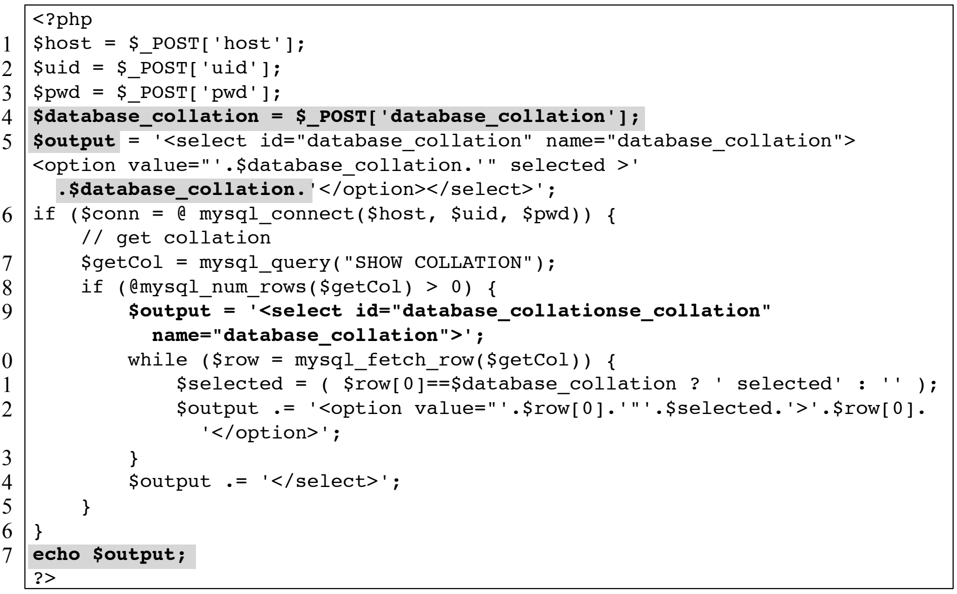
In order to illustrate our analysis, we will show an actual bug that our implementation found in the content management system MODx CMS version 1.0.3. We have reported this bug to the developers [7], who acknowledge the presence of the bug. In this example we use the PHP program in Figure 12, which was publicly available on 2010-5-4.

One of the steps of the installation process lets the user choose a database collation from a small suite of options. Users specify this database via three parameters: `host`, `uid` and `pwd`. Users also specify their choice for a collation system via a string, which the PHP program stores in the variable `database_collation`. The PHP file queries a database, using this variable as a key. However, in case the parameters `host`, `uid` or `pwd` do not determine a valid database, the module receives a collation option from a variable originated from a post request, i.e., a form. This string, stored in `database_collation`, is printed in the output without sanitization, as we see in Line 17 of Figure 12. Therefore, in order to print a malicious script in the user's webpage, we can choose an invalid host for the database, and write the script code directly in the form that feeds `database_collation`. For instance, we can steal cookies from the user's browsing environment with the string "`</option></select><script>window.alert(document.cookie); </script>`". Our analysis is able to find this vulnerability, as we illustrate in Figure 13. The reachability graph that we build for the example program contains a path from the variable `database_collation`, which is initialized from a source, to the function `echo`, which we qualify as a sink.

6. Related Work

The tainted flow problem fits the *Information Flow* framework proposed by Denning and Denning [23]. We recommend the survey of Sabelfeld and

[7] `http://www.securityfocus.com/bid/41454`



```
   <?php
 1 $host = $_POST['host'];
 2 $uid = $_POST['uid'];
 3 $pwd = $_POST['pwd'];
 4 $database_collation = $_POST['database_collation'];
 5 $output = '<select id="database_collation" name="database_collation">
   <option value="'.$database_collation.'" selected >'
     .$database_collation.'</option></select>';
 6 if ($conn = @ mysql_connect($host, $uid, $pwd)) {
       // get collation
 7     $getCol = mysql_query("SHOW COLLATION");
 8     if (@mysql_num_rows($getCol) > 0) {
 9         $output = '<select id="database_collationse_collation"
             name="database_collation">';
10         while ($row = mysql_fetch_row($getCol)) {
11             $selected = ( $row[0]==$database_collation ? ' selected' : '' );
12             $output .= '<option value="'.$row[0].'"'.$selected.'>'.$row[0].
                 '</option>';
13         }
14         $output .= '</select>';
15     }
16 }
17 echo $output;
   ?>
```

Figure 12: An installation file used in MODx CMS version 1.0.3. This file contains a XSS vulnerability, which we have highlighted in boldface.

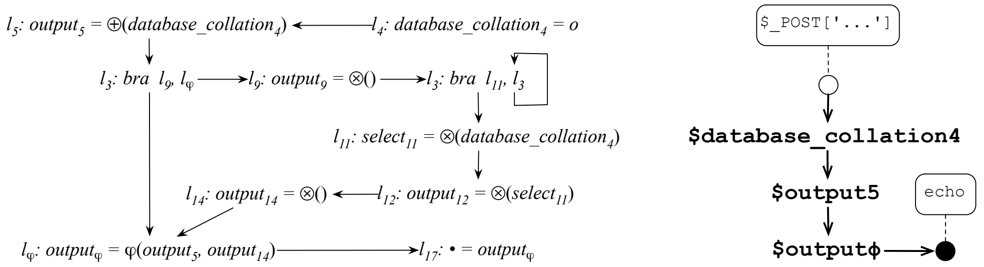

l_5: output$_5$ = ⊕(database_collation$_4$)
l_4: database_collation$_4$ = o
l_3: bra l_9, l_φ
l_9: output$_9$ = ⊗()
l_3: bra l_{11}, l_3
l_{11}: select$_{11}$ = ⊗(database_collation$_4$)
l_{14}: output$_{14}$ = ⊗()
l_{12}: output$_{12}$ = ⊗(select$_{11}$)
l_φ: output$_\varphi$ = φ(output$_5$, output$_{14}$)
l_{17}: • = output$_\varphi$
$_POST['...']
$database_collation4
$output5
echo
$outputϕ

Figure 13: (Left) the Nano-PHP representation of the program in Figure 13 – we show only the highlighted lines. (Right) The reachability graph.

Myers [24] for a more recent view about the field. The main goal of information flow analyses is to track how data flows along the many paths that constitute a program code. Early implementations of information flow algorithms relied on type systems such as Smith *et al.*'s [25] and Myers *et al.* [26]. As an illustrative example, Ørbæk and Palsberg [6]'s tainted flow analysis, from which we started the present work, is type based. However, more recent approaches to the tainted flow problem tend to use program slicing [19] and/or dataflow analyses instead of pure type systems.

The tainted flow problem is well known in the literature [4, 3, 10, 27, 28]. Wasserman and Su [4] have used context-free grammars and string analysis [29] to prove that functions manipulate strings safely. Another strategy, which uses symbolic execution to solve the tainted flow problem, was proposed by Xie and Aiken [3]. While our analysis has conditional validators powered by the e-SSA representation, these other approaches try to infer new functions as validators. However, a direct comparison between these previous two works and ours is not possible, because the tools are not publicly available. We can only speculate that, by using symbolic execution or string analysis, they are more expensive than ours, although likely more precise.

However, there exist publicly available tools that perform tainted flow analysis. Our work has found strong motivation in the design of Pixy [10], a PHP analyzer that uses the dataflow framework given in Section 3. There are also tools that rely on program dependence graphs to implement efficient tainted flow analyzers. An example is MARCO [27], a Java bytecode checker based on program slicing. TAJ [28], a tainted flow tracker for Java, has extended some of the techniques present in MARCO to an industrial level. TAJ seems to have important influence in the design of ACTARUS [5], which, on its turn, served as the baseline of F4F [30]. All these tools have been shown to scale to programs having millions of lines of code. Finally, the Joana tool [31, 32, 33] elegantly combines the dataflow monotone framework from Section 3 with the classic program dependence graph [34] in order to detect tainted flow vulnerabilities. Our most important contribution with relation to these tools is path sensitiveness, a property that we obtain from the e-SSA intermediate representation. We believe that versions of Pixy, MARCO, TAJ, ACTARUS, F4F or Joana augmented with the means to recognize e-SSA form programs will be also path-sensitive.

Many compiler analyses are based on graph reachability. In this case, the subject graph normally represents part of a *program slice* [19]. This strategy was made popular by the pioneering works of Choi *et al.* [35] and Reps *et al.* [36]. Scholz *et al.* [37] provide a clear explanation about how graphs are used to model dataflow problems. The tainted flow problem has been modeled as instances of graph reachability before [28, 31, 38]. In particular, relying on the notion of *thin slicing* [39], Tripp *et al.* [28] have been able to analyze remarkably large benchmarks. However, to the best of our knowledge, we present the first algorithm that uses the e-SSA representation to model conditional validators inside the graph reachability framework.

The e-SSA intermediate program representation [7] allows us to model a dataflow problem *sparsely* [35]. There exist many program representations that have been designed with this purpose. The most well known member of this family is the Static Single Assignment (SSA) form [17]. Another program representation that has been conceived with similar objectives is the Static Single Information (SSI) form [40, 41], which deals with backward dataflow analyses. We opted to use the e-SSA form because, contrary to SSA form, it allows us to capture information from conditional tests. The SSI representation also gives us this type of information; however, it inserts almost seven times more copies into the source program when compared to the e-SSA form and takes almost 15 times longer to build [42].

7. Conclusions

We have presented a novel technique to statically identify tainted variable attacks in web applications. We modeled our static analysis as a reachability problem on a constraint graph, which is computed from an e-SSA form program. The e-SSA intermediate program representation [7] is fundamental to the efficiency of our approach: we can find tainting scenarios very efficiently using the e-SSA constraint-graph and recent developments have shown that it is possible to compute e-SSA also very efficiently [42]. Our analysis was able to find real security vulnerabilities in popular web applications from a variety of sources. We have reported the bugs that we found to the application maintainers. Some have acknowledged and fixed the vulnerabilities.

We compared precision and efficiency of our solution with an iterative dataflow algorithm [10]. We implemented both approaches on top of `phc`, an open source PHP compiler. As for precision, the e-SSA and dataflow analyses are similar: they found exactly the same vulnerabilities. As for efficiency, the e-SSA is often faster. In our experiments, we show that our solution with e-SSA tends to become faster than its dataflow counterpart when processing larger PHP files. Our implementation of the algorithm that converts a program to e-SSA form is currently available in the `phc` compiler. The framework to detect tainted variable vulnerabilities is available at `https://github.com/rimsa/tainted-phc.git`. Important to note that although dataflow algorithms are far from a new concept, the formal definition from Section 3.1 is a contribution of our work.

Limitations. We decided to perform our tainted flow analysis on PHP programs because this language is widely used in the development of web applications. Unfortunately, PHP presents unique challenges to static analyzers [9]. To avoid the herculean task of modeling the whole language semantics, we decided to rely on a compiler infra-structure – `phc`'s – that already performs many useful analyses. As expected, our analysis is conditioned to the same limitations of the compiler [8]. So far, `phc` does not allows us to analyze statically programs showing runtime code inclusion and runtime code evaluation. The `phc` compiler deals with such features by resolving their values at compile-time, if they can be inferred statically. Otherwise, the analyzer is forced to abort due to an incomplete view of the whole program. An alternative to these deficiencies is runtime code monitoring, a technique that tracks the flow of information dynamically. Nevertheless, static approaches like ours are still useful to mitigate the amount of instrumentation that must be used to monitor programs, as it has been demonstrated by Halfond *et al.* [43], and Huang *et al.* [44].

Future Work. In the future, we plan to reduce the rate of false positives that our analysis reports. To that end, we will transform our analysis from intra- to inter-procedural. To achieve this goal, we need to modify the `phc` whole program analysis to convert the program into SSA (and in e-SSA) on demand, as described in [9, p.63]. The conversion must run alongside other client analysis, such as alias-analysis and type-inference. However, we must overcome `phc` scalability issues concerning memory management before attempting to implement the inter-procedural analysis.

Acknowledgment. Andrei Rimsa is supported by CAPES. We thank Paul Biggar for invaluable help with the `phc` compiler, plus the anonymous reviewers for helping to improve the text. This work was partially supported by the National Institute of Science and Technology for Software Engineering (INES: `www.ines.org.br`).

References

[1] D. Scott, R. Sharp, Specifying and enforcing application-level web security policies, Trans. on Knowl. and Data Eng. 15 (2003) 771–783.

[2] H. Bojinov, E. Bursztein, D. Boneh, XCS: Cross channel scripting and its impact on web applications, in: CCS, ACM, 2005, pp. 420–431.

[3] Y. Xie, A. Aiken, Static detection of security vulnerabilities in scripting languages, in: USENIX-SS, USENIX Association, 2006, pp. 179–192.

[4] G. Wassermann, Z. Su, Sound and precise analysis of web applications for injection vulnerabilities, in: PLDI, ACM, 2007, pp. 32–41.

[5] S. Guarnieri, J. Dolby, M. Pistoia, S. Teilhet, O. Tripp, R. Berg, Saving the world wide web from vulnerable javascript, in: ISSTA, ACM, 2011, pp. 177–187.

[6] P. Ørbæk, J. Palsberg, Trust in the λ-calculus, Journal of Functional Programming 7 (1997) 557–591.

[7] R. Bodik, R. Gupta, V. Sarkar, ABCD: eliminating array bounds checks on demand, in: PLDI, ACM, 2000, pp. 321–333.

[8] P. Biggar, Design and Implementation of an Ahead-of-Time Compiler for PHP, Ph.D. thesis, Trinity College Dublin, 2009.

[9] P. Biggar, E. de Vries, D. Gregg, A practical solution for scripting language compilers, in: SAC, ACM, 2009, pp. 1916–1923.

[10] N. Jovanovic, C. Kruegel, E. Kirda, Pixy: A static analysis tool for detecting web application vulnerabilities (short paper), in: S&P, IEEE, 2006, pp. 258–263.

[11] N. Jovanovic, C. Kruegel, E. Kirda, Precise alias analysis for static detection of web application vulnerabilities, in: PLAS, ACM, 2006, pp. 27–36.

[12] R. Chugh, J. A. Meister, R. Jhala, S. Lerner, Staged information flow for javascript, in: PLDI, ACM, 2009, pp. 50–62.

[13] A. Rimsa, Efficient detection of tainted flow vulnerabilities, Master's thesis, Federal University of Minas Gerais, 2010.

[14] A. V. Aho, M. S. Lam, R. Sethi, J. D. Ullman, Compilers: Principles, Techniques, and Tools (2nd Edition), Addison Wesley, 2006.

[15] A. W. Appel, J. Palsberg, Modern Compiler Implementation in Java, Cambridge University Press, 2nd edition, 2002.

[16] J. Palsberg, Efficient inference of object types, Inf. Comput. 123 (1995) 198–209.

[17] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, F. K. Zadeck, Efficiently computing static single assignment form and the control dependence graph, TOPLAS 13 (1991) 451–490.

[18] T. Lengauer, R. E. Tarjan, A fast algorithm for finding dominators in a flowgraph, TOPLAS 1 (1979) 121–141.

[19] M. Weiser, Program slicing, in: ICSE, IEEE, 1981, pp. 439–449.

[20] F. C. Chow, S. Chan, S.-M. Liu, R. Lo, M. Streich, Effective representation of aliases and indirect memory operations in SSA form, in: CC, Springer, 1996, pp. 253–267.

[21] A. Pioli, M. Burke, M. Hind, Conditional Pointer Aliasing and Constant Propagation, Technical Report 99-102, SUNY at New Paltz, 1999.

[22] A. A. Rimsa, M. d'Amorim, F. M. Q. Pereira, Efficient static checker for tainted variable attacks, in: SBLP, SBC, 2010, pp. 1–14.

[23] D. E. Denning, P. J. Denning, Certification of programs for secure information flow, Commun. ACM 20 (1977) 504–513.

[24] A. Sabelfeld, A. Myers, Language-based information-flow security, Journal on Selected Areas in Communications 21 (2003) 5–19.

[25] G. Smith, D. Volpano, Secure information flow in a multi-threaded imperative language, in: POPL, ACM, 1998, pp. 355–364.

[26] A. C. Myers, B. Liskov, Protecting privacy using the decentralized label model, TOSEM 9 (2000) 410–442.

[27] M. Pistoia, R. Flynn, L. Koved, V. Sreedhar, Interprocedural analysis for privileged code placement and tainted variable detection, in: ECOOP, Springer, 2005, pp. 362–386.

[28] O. Tripp, M. Pistoia, S. Fink, M. Sridharan, O. Weisman, TAJ: Effective taint analysis of web applications, in: PLDI, ACM, 2009, pp. 87–97.

[29] A. S. Christensen, A. Møller, M. I. Schwartzbach, Precise analysis of string expressions, in: SAS, Springer, 2003, pp. 1–18.

[30] M. Sridharan, S. Artzi, M. Pistoia, S. Guarnieri, O. Tripp, R. Berg, F4F: Taint analysis of framework-based web applications, in: OOPSLA, ACM, 2011, pp. 1053–1068.

[31] C. Hammer, J. Krinke, G. Snelting, Information flow control for java based on path conditions in dependence graphs, in: ISSSE, IEEE, 2006, pp. 1–10.

[32] C. Hammer, Information Flow Control for Java, Ph.D. thesis, Universitatsverlag Karlsruhe, 2009.

[33] C. Hammer, G. Snelting, Flow-sensitive, context-sensitive, and object-sensitive information flow control based on program dependence graphs, International Journal of Information Security 8 (2009) 399–422.

[34] J. Ferrante, K. J. Ottenstein, J. D. Warren, The program dependence graph and its use in optimization, TOPLAS 9 (1987) 319–349.

[35] J.-D. Choi, R. Cytron, J. Ferrante, Automatic construction of sparse data flow evaluation graphs, in: POPL, ACM, 1991, pp. 55–66.

[36] T. Reps, S. Horwitz, M. Sagiv, Precise interprocedural dataflow analysis via graph reachability, in: POPL, ACM, 1995, pp. 49–61.

[37] B. Scholz, C. Zhang, C. Cifuentes, User-input dependence analysis via graph reachability, Technical Report, Sun Microsystems, Inc., 2008.

[38] J. S. Foster, T. Terauchi, A. Aiken, Flow-sensitive type qualifiers, in: PLDI, ACM, 2002, pp. 1–12.

[39] M. Sridharan, S. J. Fink, R. Bodik, Thin slicing, in: PLDI, ACM, 2007, pp. 112–122.

[40] S. Ananian, The Static Single Information Form, Master's thesis, MIT, 1999.

[41] J. Singer, Static Program Analysis Based on Virtual Register Renaming, Ph.D. thesis, University of Cambridge, 2006.

[42] A. L. C. Tavares, F. M. Q. Pereira, M. A. S. Bigonha, R. Bigonha, Efficient SSI conversion, in: SBLP, pp. 1–14.

[43] W. G. J. Halfond, A. Orso, AMNESIA: analysis and monitoring for neutralizing sql-injection attacks, in: ASE, ACM, 2005, pp. 174–183.

[44] Y.-W. Huang, F. Yu, C. Hang, C.-H. Tsai, D.-T. Lee, S.-Y. Kuo, Securing web application code by static analysis and runtime protection, in: WWW, ACM, 2004, pp. 40–52.

A. Proofs of Theorems

A.1. Proof of Theorem 3.1

If $[\![l_1, l_2]\!](v) =$ tainted, then this abstract value must have been set either by Rule 1 or 3 in Figure 2, because Rules 2, 4 and 6 do not define variables, and Rule 5 always defines clean variables.

- If v has been bound to tainted by Rule 1, then we are done.
- If v has been bound to tainted by Rule 3, then we have that one of the parameters of $l : v = \oplus(x_1, \ldots, x_n)$ is tainted. The result holds by induction on (l, l_+). This propagation chain must terminate, because the program is finite. □

A.2. Proof of Theorem 4.1

⇒) We must show that $\Sigma'[v]$ = tainted only if v is influenced by some assignment from source, and v reaches an assignment to sink. We start *markTaintedVars* with an environment that maps every variable to clean. Therefore, if $\Sigma'[v] =$ tainted, then v is defined or used by an instruction that is visited, as we see in line 9 of Figure 8. If we initialize *markTaintedVars* only with source instructions, then only instructions influenced by these seeds are visited, as we see in line 12 and 13. Finally, as we see in line 21, only assignments to sink generate tainted values in Σ; the other instructions only propagate them.

⇐) We must show that if a variable v is influenced by an assignment from source $i_\circ$, and reaches an assignment to sink $i_\bullet$, then $\Sigma'[v] =$ tainted.

The proof is an induction on the size of the use-chain from the instruction i that is currently visited to $i_\bullet$. We assume that a variable $\Sigma'[v] = \text{tainted}$ if v reaches $i_\bullet$ through a use-chain of n nodes. In the base case, we have that $i = i_\bullet$, e.g., $\bullet = v$, and in line 21 of our algorithm we set $\Sigma'[v] = \text{tainted}$. In the induction step we consider lines 22-25 of our algorithm. We have that v is defined by an instruction $i \in V$, and function *doUseChainSearch* is called on v. If v reaches $i_\bullet$ through a use-chain of $n + 1$ nodes, then there must be a path between some $i' \in \mathit{USE}(v)$ and $i_\bullet$ with n nodes. If i' is an assignment to sink v is tainted in line 21. Otherwise, by induction, i' defines a variable x such that $\Sigma'[x] = \text{tainted}$, and $\mathit{taintUse}\ \Sigma'\ \mathit{USE}(v) = \textbf{true}$. In this case, we set $\Sigma'[v] = \text{tainted}$ at line 16 of Figure 8. □