Tainted Flow Analysis on e-SSA-form Programs

Andrei Rimsa[1], Marcelo d'Amorim[2], Fernando Magno Quintão Pereira[1]

[1] UFMG – 6627 Antônio Carlos Av, 31.270-010, Belo Horizonte, Brazil
[2] UFPE – Av. Prof. Luis Freire, 50.740-540, Recife, Brazil
`rimsa@live.com, damorim@cin.ufpe.br, fpereira@dcc.ufmg.br`

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 data-flow analysis. Using the same infrastructure, we compared a state-of-the-art data-flow 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 data-flow algorithm for bigger inputs. We have reported the bugs that we found, and an implementation of our algorithm is now publicly available.

1 Introduction

Web applications are pervasive in the Internet. They are broadly used and often manipulate sensitive information. It comes to no surprise that web applications are common targets of *cyber attacks* [24]. A cyber attack typically initiates with a remote attacker carefully forging inputs to corrupt a running system. A study performed by CVE[3] with statistics for the year 2006 shows that *cross-site scripting* accounts for 18.5% of the web vulnerabilities, while *PHP includes* and *SQL injection* account, respectively, for 13.1% and 13.6%. All three vulnerabilities are commonly found in web applications. To put the significance of these threats in perspective, the annual SANS's report[4] estimates that a particular type of attack – malicious SQL injection – has happened approximately 19 million times in July of 2009. Therefore, the static detection of potential vulnerabilities in web applications is an important problem.

Many web vulnerabilities are described as *Tainted Flow Attacks*. Examples include: SQL injection, cross-site scripting, malicious file inclusion, unwanted command executions, eval injections, and file system attacks [24, 29, 31]. This

[3] `http://cve.mitre.org/docs/vuln-trends/index.html`
[4] `http://www.sans.org/top-cyber-security-risks/origin.php`

pattern consists of a remote individual exploring potential leaks in the system via its public interface. In this context, the interface is the web and the vulnerability is the lack of "sanity" checks on user-provided data before using it on sensitive operations. To detect this kind of attack one needs to answer the following question: does the target program contain a path on which data flows 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 the potential threat. We call the previous question the *Tainted Flow Problem.*

The tainted flow problem was formalized by Ørbæk and Palsberg in 1997 as an instance of type-checking [16]. They wrote a type system to the 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 on the Pixy tool. This algorithm was a data-flow version of Ørbæk and Palsberg's type system. It has an average $O(V^2)$ running-time complexity, yet the Pixy's implementation suffers from worst case $O(V^4)$ complexity. Ørbæk and Palsberg's solution, when seen as a data-flow problem, admits a worst case $O(V^3)$ solution [16, p.30].

This paper improves on the complexity of these previous results. The algorithm that we propose is, in the worst case, quadratic on the number of variables in the source program, both in terms of time and space. The low asymptotic complexity is justified by the use of a program representation called *Extended Static Single Assignment* (e-SSA) form, introduced by Bodik *et al.* [5], which can be computed in linear time on the program size. This intermediate representation makes it possible to solve the tainted flow problem as a *sparse* analysis, which associates constraints directly to program variables, instead of associating them to variables at every program point. 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 4.4.
– An implementation of the algorithm on top of `phc` [3, 4], an open source PHP compiler [5]. 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 [13, 14, 31], and the consequent exposure of previously unknown vulnerabilities. See Section 5.

Our analysis can be generalized to other procedural languages. We chose PHP for two reasons. First, it is popular for developing server-side web applications. For example, PHP programs can be found in over 21 million Internet domains[6]. Second, PHP has been the focus of previous research on static detection of tainted flow vulnerabilities, and benchmarks are easily available.

[5] `http://www.phpcompiler.org/`
[6] `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 peform 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, proving that untrusted information is safe, removing malicious contents from tainted data, or firing exceptions when necessary. The literature describes many kinds of tainted flow attacks. Some noticeable examples are cross-site scripting (XSS) [9, 24], SQL injection [29, 31], malicious evaluations [7], local/remote file inclusions [8], and unwanted command execution [9]. In this section, we explain two of these vulnerabilities in more detail; however, in the rest of the paper we chose to focus on cross site scripting attacks only. Important to note that our framework is capable of handling other types of attacks. In particular, we have showed elsewhere [21] that it can be used to search for SQL injections.

2.1 Cross-Site Scripting

A cross-site scripting attack can occur when a user is able to dump HTML text 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` from the code fragment below.

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

Note that a potentially malicious JavaScript could be used instead of `does.something.evil`. A workaround for this threat is to strip HTML-related data from the user input. In this case, from the JavaScript passed as input. The function `htmlentities`, shown below, does the trick by encoding special characters into their respective HTML entities. For example, this function translates the symbol "<" to "`&lt;`".

```
<?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`

[7] `http://cwe.mitre.org/data/definitions/95.html`
[8] `http://projects.webappsec.org/Remote-File-Inclusion`
[9] `http://secunia.com/advisories/26201/`

2.2 SQL Injection Attacks

The SQL injection attack is another common type of security flaw. In this attack 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 outputs one row and therefore bypass the authentication procedure.

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 is 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 to define another variable.

Name	**Instruction**	**Example**
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();`

Fig. 1: The Nano-PHP syntax.

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

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

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

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

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

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

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

Fig. 2: Operational semantics of Nano-PHP.

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

Store Σ : $Var \rightarrow Abs$	e.g., $\{x_1 \mapsto clean,\ \ldots,\ x_n \mapsto tainted\}$
Code Heap F : $L \rightarrow [Ins]$	e.g., $\{l_1 \mapsto i_1 \ldots i_a,\ \ldots,\ l_n \mapsto i_b\}$
Instruction Sequence I : $[Ins]$	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 $\{\perp, \text{clean}, \text{tainted}\}$. The store Σ binds each variable name, say $x \in Var$, to an abstract value $v \in Abs$. 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 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

$\sqcup$	$\perp$	**clean**	**tainted**
$\perp$	$\perp$	clean	tainted
clean	clean	clean	tainted
tainted	tainted	tainted	tainted

Table 1: Definition of least upper bound over pairs of abstract values.

must be bound to clean in order to execute a safe assignment to sink. Rule S-Simple says that, given an assignment $x = \otimes(x_1, x_2, \ldots, x_n)$, the abstract state of x is defined by folding the join operation (as described on Table 1) onto the list of variables in the right hand side, e.g.: $x_1 \sqcup x_2 \ldots \sqcup x_n$. Rule S-Branch defines a non-deterministic branch choice: 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 `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. Before we move on to describe the traditional data flow solution to the tainted flow problem, a note about functions is in order. 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 algorithms in this paper can be produced by creating assignments from actual to formal parameters. We opted for not doing it due to an engineering shortcoming: our limited knowledge of `phc` has hindered us thus far from crossing the boundaries of functions.

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

Definition 1. The Tainted Flow Problem
Instance: *a Nano-PHP program P.*
Problem: *determine if the machine can get stuck executing P.*

l	$[\![\_]\!]$
$x = \circ$	$[\![l, l_+]\!] = \mathit{JOIN}(l) \setminus [x \mapsto \text{tainted}]$
$\bullet = x$	$[\![l, l_+]\!] = \mathit{JOIN}(l)$
$x = \otimes(x_1, \ldots, x_n)$	$[\![l, l_+]\!] = \mathit{JOIN}(l) \setminus [x \mapsto \mathit{JOIN}(l)(x_1) \sqcup \ldots \sqcup \mathit{JOIN}(l)(x_n)]$
`bra` $l_1, \ldots, l_n$	$[\![l, l_i]\!] = \mathit{JOIN}(l), 1 \leq i \leq n$
$x =$ `filter`	$[\![l, l_+]\!] = \mathit{JOIN}(l) \setminus [x \mapsto \text{clean}]$
`validate` x, l_c, l_t	$[\![l, l_c]\!] = \mathit{JOIN}(l) \setminus [x \mapsto \text{clean}]$ $[\![l, l_t]\!] = \mathit{JOIN}(l)$

Table 2: Data-Flow equations to solve the Tainted Flow Problem.

Data Flow Analysis. Given a Nano-PHP program, we can solve the tainted flow problem using a *forward-must* data flow analysis. Our analysis binds information to *program points*, which are the regions between pairs of consecutive Nano-PHP labels. We define a lattice $(\mathit{Abs}, <)$ by augmenting the set Abs with the following ordering $\bot <$ clean $<$ tainted. Table 1 shows the least upper bound for subsets of Abs including pairs of elements from Abs. The map lattice $(\mathit{Var} \to \mathit{Abs}, <')$ is obtained with the typical lifting of the lattice associated to Abs. Recall that the set Var is finite. We represent data-flow information with the function $[\![\_]\!] : L \to L \to \mathit{Var} \to \mathit{Abs}$. This function associates to each program point (l, l') a map storing the abstract values of each program variable. We use the notation $[\![l_1, l_2]\!]$ to denote information at (l_1, l_2). It abbreviates the function application $([\![\_]\!]l_1)l_2$. Note that $[\![\_]\!]$ is also a lattice.

Table 2 defines the transfer functions $(\mathit{Var} \to \mathit{Abs}) \to (\mathit{Var} \to \mathit{Abs})$ 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 $\mathit{PRED}(l)$ be the set of program points immediately before label l, and define the auxiliary function JOIN as follows:

$$\mathit{JOIN}(l) = \bigsqcup [\![l_i, l]\!] \;, \qquad l_i \in \mathit{PRED}(l)$$

Given two functions $[\![k', k]\!]$ and $[\![l', l]\!]$, we define $\bigsqcup \; \{[\![k', k]\!], [\![l', l]\!]\}$ as $\lambda v \,.([\![k', k]\!]v) \sqcup ([\![l', l]\!]v)$, with $\sqcup$ 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.

The join 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 JOIN over a map lattice. Informally, the semantics of this operation is to apply $\sqcup$ over elements on the image of the functions according to the definition on Table 1. For example $\{x \mapsto \mathit{clean}, y \mapsto \mathit{clean}\} \sqcup \{x \mapsto \mathit{tainted}, y \mapsto \bot\} = \{x \mapsto \mathit{clean} \sqcup \mathit{tainted}, y \mapsto \mathit{clean} \sqcup \bot\}$.

```
<?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);
}
?>
```

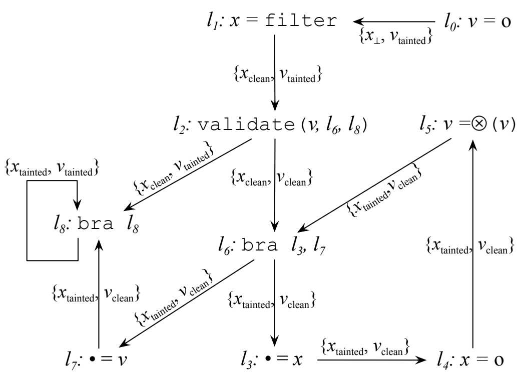


Fig. 3: A simple PHP program (left), and its equivalent Nano-PHP version (right), augmented with the result of data-flow analysis.

Illustrative Example. Figure 3 illustrates the result of a data-flow 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 $v = \otimes(v)$. We use l_8, a label jumping to itself, to mark the end of the program. We show the maps produced by the data-flow analysis on the edges of the Nano-PHP program. In this program the data-flow 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`.

Complexity. We can solve this data-flow analysis using the chaotic iteration model. 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 data-flow 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 [2]. In particular, Palsberg [17] gives an $O(V^3)$ type-inference algorithm that solves the tainted flow problem. In practice, this data-flow analysis is $O(V \times I)$ [2, p.209].

4 The Proposed Solution

In this section we describe our solution to the tainted flow problem. Our approach is divided into the three parts below. We give time complexity in terms of the number of variables (V) in the source program.

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 α is the inverse Auckerman function, normally regarded as constant, and the insertion of ϕ-functions is $O(V^2)$, yet linear in practice [2, 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 the Linchpin of Fast 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 [10]. This representation has been used by Bodik *et al.* [5] 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 [2, 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.* [10] 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 reach 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 [15]. E-SSA-form programs are also SSA-form programs; thus, they have the property that each variable has only one definition.

Second, we modify the syntax of the validator instruction, which become `validator` (x, x_c, l_c, x_t, l_t) [10]. Conceptually, the validator splits the live range of

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

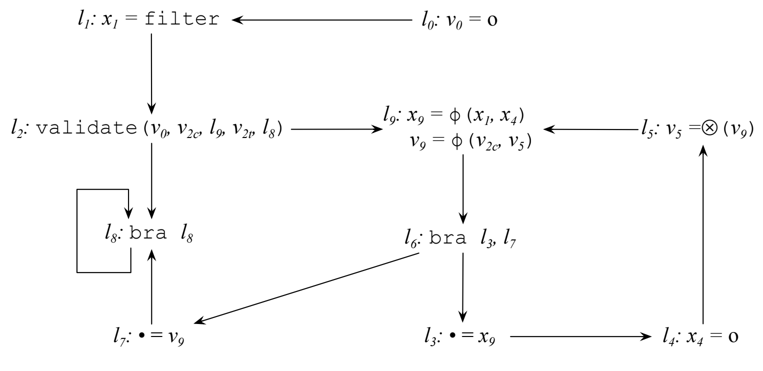
variable x in two parts, depending on whether or not its abstract value is tainted. Note that when converting a program into e-SSA form, we rename every use of x in labels dominated by l_c to x_c, and rename every use of x in labels dominated by l_t to x_t. The new instruction has the following semantics:

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

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

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.

The e-SSA representation allows us to acquire static information from the outcome of conditionals. Hence, we can associate unique constraints to variables, as Figure 4 illustrates. 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 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.



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: • = v_9
l_3: • = x_9
l_4: x_4 = o

Fig. 4: The example of Figure 3 converted into e-SSA form.

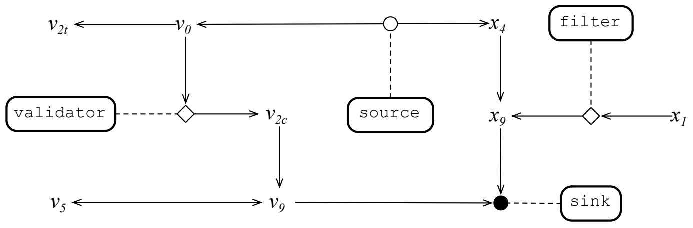


Fig. 5: The reachability graph built after the program in Figure 4.

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.

Definition 2 rephrases the tainted flow problem as an instance of graph reachability. The traversal of the reachability graph is related to the notion of program slicing [30]. Any node u that reaches a node v is part of the program slice that defines the behavior of v.

Definition 2. 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 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 (Left). Assuming that `$_GET` is a source and `echo` is a sink, then the program is logically bug free.

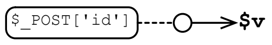
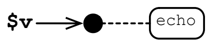
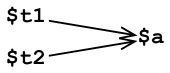
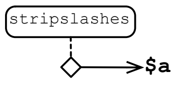
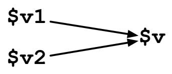
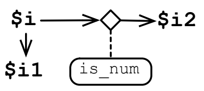
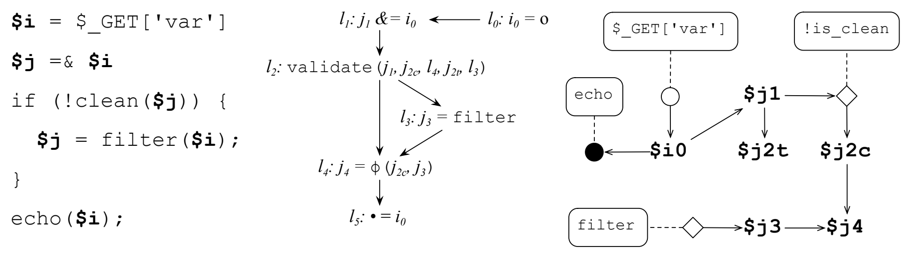
Instruction	Example	Nodes
$v = \circ$	`$v = $_POST['id']`	
$\bullet = v$	`echo($v)`	
$v = \otimes(v_1, \ldots, v_2)$	`$a = $t1 * $t2`	
$v =$ `filter`	`$a = stripslashes($t1)`	
$v = \phi(v_1, \ldots, v_2)$	`$v = phi($v1, $v2)`	
`validate` (v, v_c, l_c, v_t, l_t)	`if(is_num($i))`	

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



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

Fig. 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.

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*

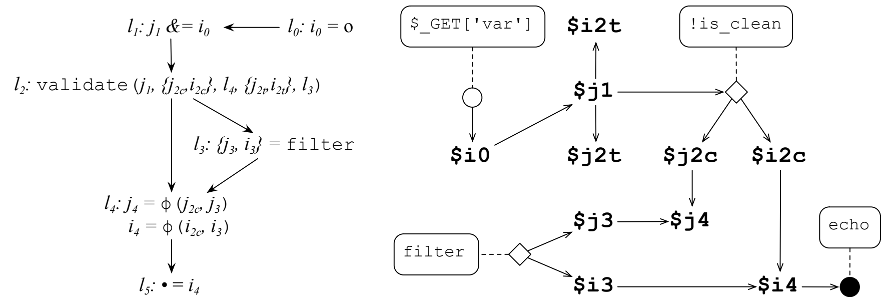


Fig. 7: (Left) input program in e-SSA form augmented with the results of alias analyses. (Right) final reachability graph.

Assignment (HSSA) form [7]. This last program representation is used internally by `phc` [3, 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 [18] 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.

4.4 A Solution Quadratic in Time and Space

The function *markTaintedVars*, given in Figure 8 finds bugs in e-SSA-form Nano-PHP programs. We use SML/NJ's syntax plus Erlang-style guards in pattern matching, as in the auxiliary function *hasTaintedChild*. This function simulates a traversal of the reachability graph that we described in Section 4.2, but it does not really build the graph. Instead, it relies on the *use-chains* of the variables to guide the traversal. The use-chain of a variable x is a function *USE* that maps x to every instruction where this variable is used.

Function *markTaintedVars* receives three parameters: a set $\{i, i_1, \ldots, i_n\}$ of instructions to process, an environment Σ that maps variables to either clean or tainted, and a set of visited instructions, which we keep to avoid visiting the same instruction twice. *MarkTaintedVars* processes each instruction forwardly, i.e, an instruction that defines a variable x is buggy if any of the instructions that use x is buggy. We assume that every variable used in a sink function is buggy. We use the auxiliary function *hasTaintedChild* to check if any of the instructions in the use chain of a variable x defines a variable that has been set as tainted in

```
fun hasTaintedChild  $\_$ $\{\ldots,(\bullet = x),\ldots\}$ $\Rightarrow$ true
  | hasTaintedChild $\Sigma$ $\{\ldots,(x = \otimes(\ldots)),\ldots\}$ $\wedge$ $\Sigma \vdash x = \text{clean}$ $\Rightarrow$ true
  | hasTaintedChild $\Sigma$ $\{\ldots,(x = \phi(\ldots)),\ldots\}$ $\wedge$ $\Sigma \vdash x = \text{clean}$ $\Rightarrow$ true
  | hasTaintedChild $\Sigma$ $\{\ldots,(\texttt{validate}(\_,\_,\_,x,\_)),\ldots\}$ $\wedge$ $\Sigma \vdash x = \text{clean}$ $\Rightarrow$ true
  | hasTaintedChild $\_$ $\_$ $\Rightarrow$ 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 hasTaintedChild $\Sigma'$ $USE(v)$
            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
```

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

the environment. Notice that neither *markTaintedVars* or *hasTaintedChild* deals with switches or filter instructions. These instructions will never define or use tainted variables, and will never be found by any of these functions.

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 data-flow analysis discussed in Section 3 and our e-SSA based analysis from Section 4 on top of the `phc` open source compiler [3, 4]. This compiler, started in 2005 by Edsko de Vries and John Gilbert, is implemented in C++, and currently uses our implementation of e-SSA as an internal representation. Our implementation of data-flow analysis uses a standard working list algorithm, and runs on a quasi-topological ordering of the CFG of the input program [2, 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 appear in previous works [13, 14, 31]. 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 happen before we have the chance to run the tainted flow analyses. A detailed account of each `phc` failure is provided by Rimsa [21].

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 data-flow analysis (Section 3) and the time to run our sparse analysis (Section 4). We run the data-flow 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, π and ϕ-functions [5, 10]. Figure 9 shows that the e-SSA based approach is faster than the data-flow 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 data-flow approach. We measure the time to analyze each function 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 1,122 function above this threshold. The largest function that we have analyzed contains 1,141 instructions. We speculate that once we cross the boundaries of functions, and analyze whole PHP applications, which might contain thousands of functions, and millions of lines of code, our analysis will be much more efficient than the data-flow approach.

Precision: Both our e-SSA based analysis and the data-flow 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,

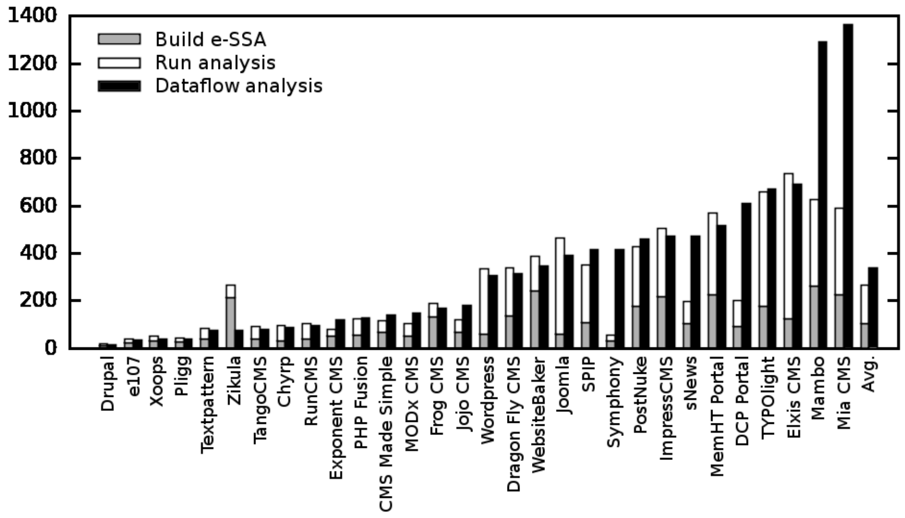


Fig. 9: Average execution time (ms) per benchmark for data-flow and e-SSA-based analyses. Bars are sorted by the time to run the data-flow based analysis.

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 CMS	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.

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 [21, 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 [11], who acknowledge the presence

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

of the bug. In this example we use the PHP program in Figure 10, 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 10. 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 finds this vulnerability, as we illustrate in Figure 11. 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.

```
    <?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;
    ?>
```

Fig. 10: An installation file used in MODx CMS version 1.0.3. This file contains a XSS vulnerability, which we have highlighted in boldface.

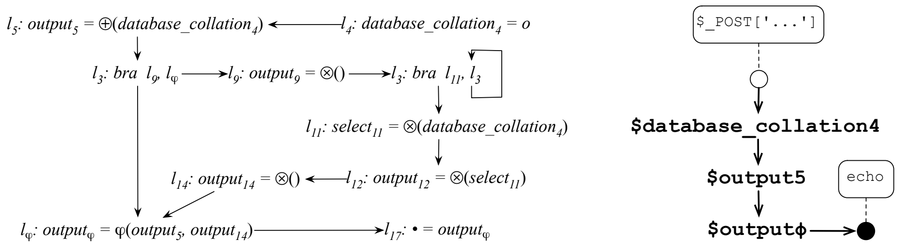

l5: output5 = ⊕(database_collation4)
l4: database_collation4 = o
l3: bra l9, lφ
l9: output9 = ⊗()
l3: bra l11, l3
l11: select11 = ⊗(database_collation4)
l14: output14 = ⊗()
l12: output12 = ⊗(select11)
lφ: outputφ = φ(output5, output14)
l17: • = outputφ
$_POST['...']
$database_collation4
$output5
echo
$outputϕ

Fig. 11: (Left) the Nano-PHP representation of the program in Figure 11 – we show only the highlighted lines. (Right) The reachability graph.

6 Related Work

The tainted flow problem is well known in the literature [13, 19, 28, 29, 31]. Wasserman and Su [29] have used context-free grammars and string analysis [8] 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 [31]. 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. There exist; however, publicly available tools that perform tainted flow analysis. One of them is MARCO [19], a Java bytecode analyzer. Another is Pixy [13], a PHP analyzer. MARCO relies on program slicing [30] to find the set of tainted variables, whereas Pixy uses a variation of the data-flow analysis from Section 3. Neither tool takes the results of conditional tests into consideration; hence, both are path insensitive – a problem that our intermediate representation permits us to circumvent.

Many compiler analyses are based on the notion of graph reachability. In this case, the subject graph normally represents part of a *program slice* [30]. This strategy was made popular by the pioneering works of Choi *et al.* [6] and Reps *et al.* [20]. For a clear explanation of the use of graphs to model data-flow problems, we recommend the work of Scholz *et al.* [23]. The tainted flow problem has been modeled as instances of graph reachability before [11, 12, 28]. In particular, relying on a modified notion of *thin slicing* [26], 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 handle conditional validators inside the graph reachability framework. Conditional validators increase the precision of our analysis, as a given variable might be treated as clean in some program path, and tainted in others, and the e-SSA representation makes it possible to model this flow sensitivity sparsely.

The e-SSA intermediate program representation [5] allows us to model a data-flow problem *sparsely.* 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 [10]. Another program representation that has been conceived with similar objectives is the Static Single Information (SSI) form [1, 25], which deals with backward data-flow 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 [27].

7 Conclusion

This paper presented a novel and efficient approach to statically identify security vulnerabilities in code that can result in tainted flow attacks. Key to our speedup was the e-SSA program representation. This enabled us to encode our analysis as a graph reachability problem using a non-iterative data flow algorithm . We have implemented our analysis on top of `phc`, an open source PHP compiler, and have used it to find real bugs in well known web applications. We reported all the new bugs that we found to the maintainers of the target applications. Some of these developers acknowledged and fixed the vulnerabilities. Our implementation of the e-SSA representation is currently available in the `phc` compiler, our analysis code is available at http://www.dcc.ufmg.br/llp/projects/phc-tainted/.

Acknowledgment. Andrei Rimsa is supported by CAPES. We thank Paul Biggar for invaluable help with the `phc` compiler, and Roberto Bigonha plus the anonymous reviewers for helping to improve the text.

References

1. Ananian, S.: The Static Single Information Form. Master's thesis, MIT (September 1999)
2. Appel, A.W., Palsberg, J.: Modern Compiler Implementation in Java. Cambridge University Press, 2nd edn. (2002)
3. Biggar, P.: Design and Implementation of an Ahead-of-Time Compiler for PHP. Ph.D. thesis, Trinity College Dublin (2009)
4. Biggar, P., de Vries, E., Gregg, D.: A practical solution for scripting language compilers. In: SAC. pp. 1916–1923. ACM (2009)
5. Bodik, R., Gupta, R., Sarkar, V.: ABCD: eliminating array bounds checks on demand. In: PLDI. pp. 321–333. ACM (2000)
6. Choi, J.D., Cytron, R., Ferrante, J.: Automatic construction of sparse data flow evaluation graphs. In: POPL. pp. 55–66 (1991)
7. Chow, F.C., Chan, S., Liu, S.M., Lo, R., Streich, M.: Effective representation of aliases and indirect memory operations in SSA form. In: CC. pp. 253–267. Springer (1996)

8. Christensen, A.S., Møller, A., Schwartzbach, M.I.: Precise analysis of string expressions. In: SAS. pp. 1–18. Springer (2003)
9. Chugh, R., Meister, J.A., Jhala, R., Lerner, S.: Staged information flow for javascript. In: PLDI. pp. 50–62. ACM (2009)
10. Cytron, R., Ferrante, J., Rosen, B.K., Wegman, M.N., Zadeck, F.K.: Efficiently computing static single assignment form and the control dependence graph. TOPLAS 13(4), 451–490 (1991)
11. Foster, J.S., Terauchi, T., Aiken, A.: Flow-sensitive type qualifiers. In: PLDI. pp. 1–12. ACM (2002)
12. Hammer, C., Krinke, J., Snelting, G.: Information flow control for java based on path conditions in dependence graphs. In: ISSSE. pp. 1–10. IEEE (2006)
13. Jovanovic, N., Kruegel, C., Kirda, E.: Pixy: A static analysis tool for detecting web application vulnerabilities (short paper). In: S&P. pp. 258–263. IEEE (2006)
14. Jovanovic, N., Kruegel, C., Kirda, E.: Precise alias analysis for static detection of web application vulnerabilities. In: PLAS. pp. 27–36. ACM (2006)
15. Lengauer, T., Tarjan, R.E.: A fast algorithm for finding dominators in a flowgraph. TOPLAS 1(1), 121–141 (1979)
16. Ørbæk, P., Palsberg, J.: Trust in the λ-calculus. Journal of Functional Programming 7(6), 557–591 (1997)
17. Palsberg, J.: Efficient inference of object types. Inf. Comput. 123(2), 198–209 (1995)
18. Pioli, A., Burke, M., Hind, M.: Conditional pointer aliasing and constant propagation. Tech. Rep. 99-102, SUNY at New Paltz (1999)
19. Pistoia, M., Flynn, R., Koved, L., Sreedhar, V.: Interprocedural analysis for privileged code placement and tainted variable detection. In: ECOOP. pp. 362–386 (2005)
20. Reps, T., Horwitz, S., Sagiv, M.: Precise interprocedural dataflow analysis via graph reachability. In: POPL. pp. 49–61. ACM (1995)
21. Rimsa, A.: Efficient detection of tainted flow vulnerabilities. Master's thesis, Federal University of Minas Gerais (UFMG) (December 2010)
22. Rimsa, A.A., d'Amorim, M., Pereira, F.M.Q.: Efficient static checker for tainted variable attacks. In: SBLP. SBC (2010)
23. Scholz, B., Zhang, C., Cifuentes, C.: User-input dependence analysis via graph reachability. Tech. rep., Sun Microsystems, Inc. (2008)
24. Scott, D., Sharp, R.: Specifying and enforcing application-level web security policies. Trans. on Knowl. and Data Eng. 15, 771–783 (2003)
25. Singer, J.: Static Program Analysis Based on Virtual Register Renaming. Ph.D. thesis, University of Cambridge (2006)
26. Sridharan, M., Fink, S.J., Bodik, R.: Thin slicing. In: PLDI. pp. 112–122. ACM (2007)
27. Tavares, A.L.C., Pereira, F.M.Q., Bigonha, M.A.S., Bigonha, R.: Efficient SSI conversion. In: Brazilian Symposium on Programming Languages (SBLP). pp. 1–14 (2010)
28. Tripp, O., Pistoia, M., Fink, S., Sridharan, M., Weisman, O.: TAJ: Effective taint analysis of web applications. In: PLDI. pp. 87–97. ACM (2009)
29. Wassermann, G., Su, Z.: Sound and precise analysis of web applications for injection vulnerabilities. In: PLDI. pp. 32–41. ACM (2007)
30. Weiser, M.: Program slicing. In: ICSE. pp. 439–449. IEEE (1981)
31. Xie, Y., Aiken, A.: Static detection of security vulnerabilities in scripting languages. In: USENIX-SS. USENIX Association (2006)