Efficient Static Checker for Tainted Variable Attacks

Andrei Rimsa[1], **Marcelo d'Amorim**[2], **Fernando M. Q. Pereira**[1]

[1]Departamento de Ciência da Computação – UFMG
Av. Antônio Carlos, 6627 – 31.270-010 – Belo Horizonte – Brazil

[2]Centro de Informática – UFPE
Av. Prof. Luis Freire – 50.740-540 – Recife – Brazil

{rimsa,fpereira}@dcc.ufmg.br,damorim@cin.ufpe.br

Abstract. *Tainted variable attacks, common in server-side scripting languages, such as PHP, originate from program inputs maliciously crafted to exploit software vulnerabilities. These vulnerabilities can be detected via data-flow analyses, which iteratively compute the abstract state ("tainted" or "clean") associated to every program variable at each program point. Existing algorithms for this problem have a worst-case cubic time on the number of program variables – an issue for large programs. This paper presents a quadratic-time algorithm to tackle the tainted flow problem. We have implemented our analysis on top of phc, an open source PHP compiler, and have scanned over 13 thousand PHP files, obtaining 130 warnings, out of which 41 are actual security vulnerabilities.*

1. Introduction

Web applications are pervasive in the Internet. They permeate sites as diverse as facebook, google and blogger, are broadly used, and often manipulate sensitive information. It comes to no surprise that web applications are common targets of *cyber attacks*. A cyber attack typically initiates with a remote attacker carefully forging inputs to corrupt a running system. A study performed in late 2004 (involving sites considered popular according to the Alexa index[1]) reports a remarkably high number of 670 security-related software bugs [Xie and Aiken 2006]. This number represents an 81% increase compared to the same period from the previous year. To put the significance of these threats in perspective, the annual SANS's report[2] estimates that a particular type of attack – malicious SQL injection – has happened approximately 19 million times in 2009 only. Static detection of potential vulnerabilities in web applications is therefore an important problem.

The *Tainted Variable Attack* is a system interaction pattern general enough to characterize different kinds of concrete (real) attacks. Examples include: cross-site scripting, SQL injection, local/remote file inclusion, unwanted command executions, malicious evaluations, and file system attacks [Xie and Aiken 2006, Wassermann and Su 2007]. The 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 leak 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

[1]http://www.alexa.com/topsites
[2]http://www.sans.org/top-cyber-security-risks/

contains a path on which data flows from some input to a sensitive place without going through a sanitizer? 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*.

This paper presents a novel and simple approach to tackle this problem. The algorithm that we propose is, in the worst case, quadratic on the number of variables in the source program. This complexity improves on previous approaches [Jovanovic et al. 2006a, Scholz et al. 2008], which have a cubic worst case. 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* [Bodik et al. 2000], which can be computed in linear time on the program size. This intermediate representation makes it possible to associate 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. Two distinguishing features of the algorithm are (i) the use of the e-SSA representation to generate constraints and (ii) an on-demand constraint solving algorithm. See Section 3.
- An implementation of the algorithm on top of phc [Biggar et al. 2009], an open source PHP compiler.
- An evaluation of the proposed approach on public PHP applications, including benchmarks used in previous works [Jovanovic et al. 2006a, Xie and Aiken 2006]. See Section 4.

Our analysis solves a general problem and can be generalized to other procedural languages. We chose PHP because it is popular for developing server-side web applications. For example, PHP programs can be found in over 21 million Internet domains[3]. Another reason was easy access to existing benchmarks.

2. Background

A *tainted variable attack* consists of a remote individual maliciously crafting program inputs to explore unknown, but predictable, vulnerabilities of a running application. This paper tackles the problem of statically finding such vulnerabilities. This problem is best known as the *Tainted Flow Problem*.

Definition 2.1. THE TAINTED FLOW PROBLEM

***Instance**: A tuple $T = (P, SO, SI, SA)$, such that:*

- *P is the subject program.*
- *SO is a set of input functions, referred to as sources.*
- *SI is a set of security-sensitive operations, referred to as sinks.*
- *SA is a set of functions that validate inputs, referred to as sanitizers.*

***Problem**: Determine if program P can make a call to a sink function $si \in SI$ passing a value v, which has been generated by a source function $so \in SO$, and that has not been sanitized by any function $sa \in SA$.*

The literature describes a number of different tainted variable attacks. Some noticeable examples are cross-site scripting (XSS) [Wassermann and Su 2008], SQL injection attacks [Xie and Aiken 2006, Wassermann and Su 2007], malicious evaluations [4], lo-

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

[4] `http://cwe.mitre.org/data/definitions/95.html`

cal/remote file inclusions [5], and command execution [6]. We describe cross-site scripting in this section. The other types of attacks follow a similar pattern of operation.

2.1. Cross-Site Scripting

Cross-site scripting typically occurs when a user is able to inject html code within a dynamically generated page. An attacker uses this vulnerability to execute malicious JavaScript code on a victim's machine. For example, one can inject in a URL a script to steal sensitive cookie information that can provide session privileges to the attacker.

Example. the program below exemplifies the cross-site scripting bug:

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

A user of the page above is able to output any given text, including html commands. Consider the scenario where execution reads from input and assigns the following data to variable `$name`: "`<script>does.something.evil;</script>`". A potentially malicious JavaScript program might be executed in the client side of the application.

A workaround for this bug is to strip malicious html contents from the user input. The function `htmlentities` does the trick by encoding special characters into their respective html entities. For example, "<" gets translated to "`&lt;`".

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

Cross-site scripting is an instance of a tainted flow attack. A possible input configuration is as follows.

Sources : PHP Superglobals, e.g `$_GET[]`, `$_POST[]`, etc.
Sinks : functions that output data to the page, such as `echo, print, printf`.
Sanitizers : `htmlentities, htmlspecialchars, strip_tags`, as well as type inference functions, such as `is_numeric`.

3. The Proposed Solution

We propose a novel algorithm to solve the tainted flow problem. Our solution receives four inputs:

1. the program for analysis;
2. a list of source functions;
3. a list of target functions;
4. a list of sanitizers.

For the sake of simplicity, we will write input programs in Nano-PHP, a reduced version of PHP, described in Section 3.1. Our analysis proceeds in two steps. First, it converts the input program into e-SSA form, using the technique described in Section 3.2. The e-SSA form allows us to characterize the tainted flow problem as a graph reachability question. In the next step, it traverses such graph to find vulnerable paths from sources to sinks. Section 3.3 shows that one can find the same solution without building the graph explicitly. In this case, we perform the traversal on the program structure.

[5] `http://projects.webappsec.org/Remote-File-Inclusion`
[6] `http://projects.webappsec.org/OS-Commanding`

3.1. Nano-PHP

We define the core language Nano-PHP to illustrate our approach. Our language has seven basic types of instructions:

Name	Instruction	Example
Source assignment	$v = s,\ v \in SO$	`$v = $_POST['content']`
Sink assignment	$s = v, s \in SI$	`echo($v)`
Aliasing set up	$v_1 = \&v_2$	`$v1 = & $v2`
Simple assignment	$v = s(v_1, \ldots, v_n)$	`$a = $t1 * $t2`
Filter	$v_1 = f(v_2), f \in SA$	`$a = htmlentities($t1)`
Validation	if $(g(v))$ then $v_1 = \sigma(v); L_1$ else $v_2 = \sigma(v); L_2, g \in SA$	`if (!is_numeric($t1)) abort();`
ϕ-function	$v = \phi(v_1, v_2)$	`$v = phi($v1, $v2)`

Figure 1 (Left) contains an example of a Nano-PHP program. The instruction in line 1 is an assignment from input. Nano-PHP separates sanitizers in two categories: filters and validators. The instruction in line 3 is an assignment from a filter. A filter, in this case, could be, for instance, the standard PHP function `htmlentities`. The conditional in line 2 is a validator, which could be, for instance, the standard function `is_numeric`. Finally, in line 5 we have a sink.

3.2. Converting the Input Program to e-SSA Form

We use the *Extended Static Single Assignment* (e-SSA) representation to simplify our tainted variable analysis. E-SSA, a superset of the well known *Static Single Assignment* (SSA) form [Cytron et al. 1991], is not a new concept. This representation has been used, for instance, to eliminate array bound checks [Bodik et al. 2000] and to perform bitwidth analysis [Stephenson et al. 2000]. There are different flavors of e-SSA form, and we use the version described by Tavares *et al* [Tavares et al. 2010], because its creation requires a very small amount of change into the input program. The main property that we use from this version of e-SSA form is called *single upward-exposed-conditional*. This property says that if variable v is used at a branch instruction i, then from i it is possible to reach at most one use of v without passing across another use.

The e-SSA form guarantees this property via a special instruction called σ-function, which split the *live ranges* of variables used in conditionals. The live range of a variable v is the collection of program points where v is alive, that is, from where it is possible to reach a statement where v is used. For instance, in Figure 1 (Middle), the σ function splits the live range of `$i0` into two new variables, `$i1` and `$i2`. The former is alive only in the regions where the conditional evaluates to false. The latter is alive in the regions where the conditional evaluates to true. We use ϕ-functions, traditionally seen in SSA-form, to join the live ranges of variables. Going back to Figure 1 (Middle), the ϕ-function in the last basic block joins the live ranges of `$i1` and `$i3`, producing `$i4`.

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 1 illustrates. The original program, in Figure 1 (Left), has one variable named `$i`. We know that this variable is clean in some program points, but not all. The e-SSA representation allows us to identify these program points precisely. The modified program

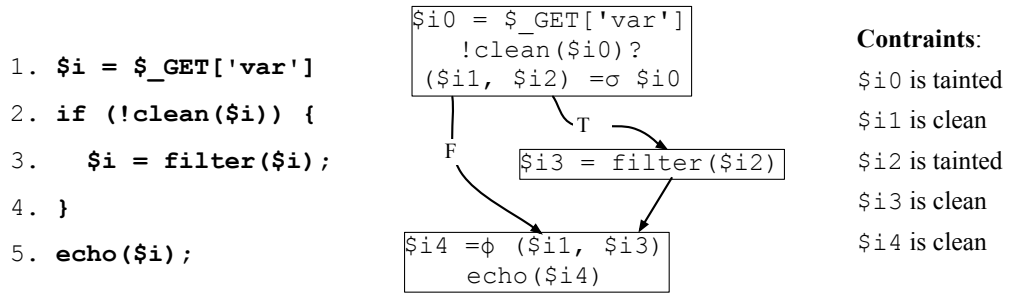


Figure 1. Using e-SSA to bind constraints to variables.

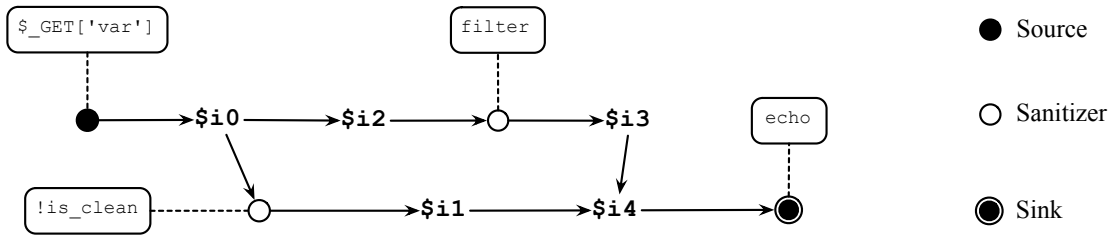


Figure 2. The reachability graph built for the program in Figure 1.

has five variables, `$i0`, `$i1`, `$i2`, `$i3` and `$i4`. Given that `$i0` comes from a source function, we let it be tainted. We know that `$i1` is clean, because it is produced by an operation of validation. That is, a Nano-PHP validation instruction, defined in Section 3.1 always produces two new variables, via a σ-function. According to the semantics of validation, one of this variables *must* be clean, and the other *must* be tainted. Thus, we know that `$i2` is tainted. However, `$i3` is produced by the application of a filter onto `$i2`; hence, `$i3` is clean. `$i4` is also clean, because it is formed by the combination of two clean variables, `$i1` and `$i3`. By associating constraints directly to variables, we obtain a *sparse analysis*. That is, we free ourselves from the need to keep information related to program points, such as "variable v is clean at program point x, but is tainted at program point y". An e-SSA variable can only be clean or tainted, but not both.

3.3. Tainted Analysis as Graph Reachability

Given a function F, we represent it as a graph G, in which each node $n_v \in G$ represents a variable $v \in F$. There is an edge linking n_u to n_v if and only if information flows from variable u to v, that is, if F contains an operation that uses u and defines v. The e-SSA form program from Figure 1 gives origin to the constraint graph in Figure 2.

The tainted flow problem in a function F is equivalent to the problem of finding a path, in the constraint graph G built after F, linking a source to a sink that does not contain any sanitizer. The graph in Figure 2 does not contain such a path; thus, we report that the underlying program has no security vulnerability. The traversal of the constraint graph gives us further knowledge about the program. For instance, we know that `$i4` is clean, because every path from a source to this variable will go across a sanitizer.

We build the constraint graph directly from the intermediate program representa-

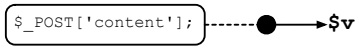
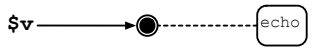
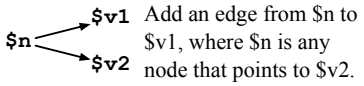
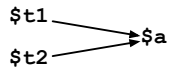
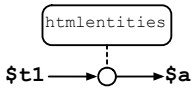
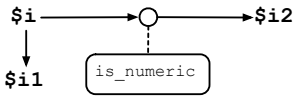
Instruction	Example	Nodes
Source assignment	`$v = $_POST['content']`	
Sink assignment	`echo($v)`	
Aliasing set-up	`$v1 = & $v2`	
Simple assignment	`$a = $t1 * $t2`	
Filter	`$a = htmlentities($t1)`	
ϕ-function	`$v = phi($v1, $v2)`	
Validation	`if(is_numeric($i))`	

Table 1. Mapping program instructions to nodes in the constraint graph.

tion. Each particular type of instruction produces a specific configuration of nodes in the constraint graph, as Table 1 shows. Once we have the constraint graph, the algorithm in Figure 3 produces a list with the vulnerable paths that the program contains. The traversal of the constraint graph is related to the notion of program slicing [Weiser 1981]. Any node u that reaches a node v is part of the program slice that defines the behavior of v.

3.4. Handling Aliasing via Duplication of Variables

Aliasing is a phenomenon typical of imperative languages, in which two names denote the same memory location. Aliasing may complicate 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 4 (Left). Assuming, as in Figure 1, 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 `$j` represent the same variable. The ordinary e-SSA representation will not catch this subtlety, as Figure 4 shows. There is a clear path from `$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 [Chow et al. 1996]. This last program representation is used internally by phc, our baseline compiler. For each assignment $v = E$ in a SSA-form program, the equivalent

```
warnings=[]; wlist=[ [n] | n <- Sources]; // n is a source node
while wlist != []
  seq = head(wlist); n = head(seq); wlist = tail(wlist);
  foreach s in succ(n)
    if (isSanitizer(s)/*clean*/ | isVisited(s)/*checked*/)
      continue;
    else if (isSource(s)) wlist = (s::seq)::wlist
    else if (isSink(s)) warnings = reverse(s::seq)::warnings
    else wlist = (s::n::seq)::wlist
report warnings
```

Figure 3. Depth-first graph traversal using functional lists.

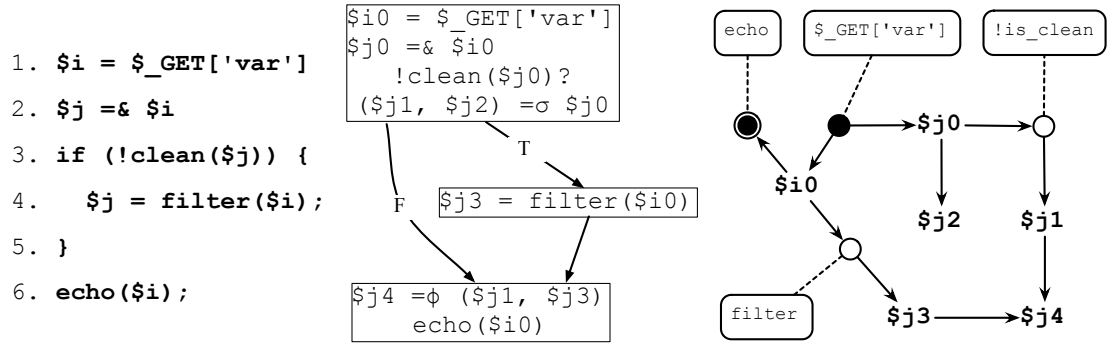


Figure 4. An example of how aliasing complicates the tainted flow analysis. In the right side we show the constraint graph built for the e-SSA form program.

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 representation contains extra assignments for each σ-function that would exist in an ordinary e-SSA form program. Every time we build a new σ-function for variable v, at program point p, we create a σ-function for each variable that is an alias of v and is alive at p.

The literature contains a plethora of methods to conservatively estimate the set of aliases of a variable. In our implementation we use the context sensitive, interprocedural alias analysis [Pioli et al. 1999] that we obtain from phc. However, it is important to emphasize that our solution to the tainted flow problem is orthogonal to the alias analysis algorithm. We could have used any other points-to solver, be it inter or intra-procedural, context sensitive or not, flow sensitive or not. We could even have not used any alias analysis at all, in this case trading precision for efficiency.

Moving on with our example, Figure 5 shows the program and the constraint graph after augmenting the original e-SSA form program with the results of alias analysis. In the new constraint graph there is no path from a source to a sink that does not go across a sanitizer. Thus, we can prove that the program is bug-free.

3.5. A Data-Flow Algorithm to solve the Tainted Flow Problem

This section describes an optimized version of our algorithm that does not build the constraint graph explicitly. The algorithm operates in two steps: (i) constraint collection, and

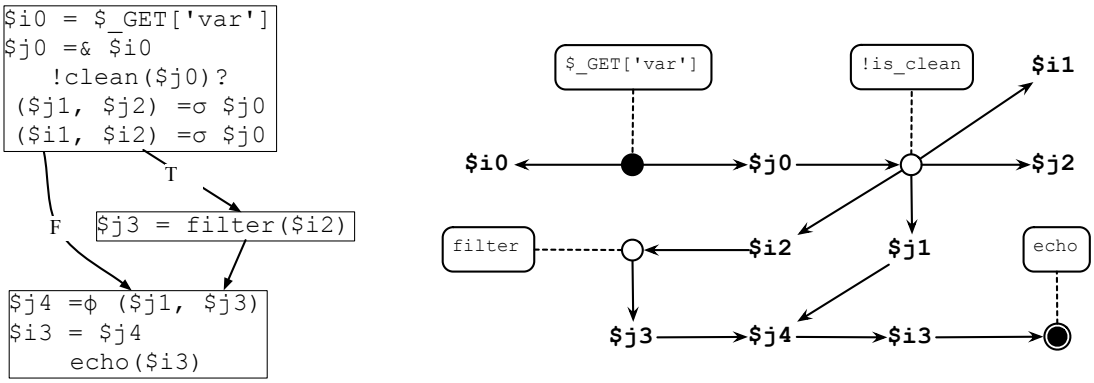


Figure 5. (Left) input program in e-SSA form augmented with the results of alias analyses. (Right) final constraint graph.

(ii) constraint solving. In the first step, a procedure visits every instruction of the program in order to build a *partial abstract state*. In the second step, a procedure completes the abstract state with contextual information. Figures 6 and 7 show respectively the constraint collection and solving algorithms.

State Representation. We use the datatype `Abs` to characterize the abstract values of our language. The value `C` corresponds to clean state, `T` to tainted state, the value `?` corresponds to unknown which can arise from undefined variables or function calls (which we do not currently handle), and value `F` refers to a future value that we will discuss shortly. We let the function `Env` to encode states, associating abstract values to variables.

Aliasing. The augmented e-SSA form discussed in Section 3.4 makes explicit the effect of aliasing. As result, it is unnecessary to model pointers at this stage. Conceptually, we use copy of value to model copy of reference: the program representation we use assures simultaneous update of all variables in the same points-to set. For example, consider the program fragment `a =& b; a = 5;`. Instead of encoding the final abstract state with separate environment and store, say with environment $\{a \mapsto 1, b \mapsto 1\}$ and store $\{1 \mapsto C\}$ as typical in dynamic semantics definitions [Pierce 2004], we collapse environment and store. In this case, we encode state as $\{a \mapsto C, b \mapsto C\}$ because the augmented e-SSA representation ensures that the updates on `a` and `b` are consistent.

Lazyness. Unfortunately, it is not possible to find all potential vulnerabilities in a single pass on the program. The reason is that, in order to define the state of a variable v, we need the state of the variables used in the instruction $v = E(u_1, \ldots, u_n)$ that defines v. This requirement forces us to visit the instructions that define each u_i before reaching the instruction that defines v. If we use the dominance tree [Lengauer and Tarjan 1979] of the target program to guide the order in which we visit the instructions, we ensure this property for every type of instruction, except ϕ-functions [Budimlic et al. 2002]. Consider the PHP program `$x=10; while(...){$x=$x+2;...}` and its SSA variant `x1=10; while(...) {x3=phi(x1,x2); x2=x3+2;...}`. There is no systematic way to visit the program instructions which ensures that we will visit the definition of `x1` and `x2` before reaching the ϕ-function [Budimlic et al. 2002]. To deal with such "future" value dependencies (possibly recursive), we create the special value `Future`. The value of vari-

able x3, for example, is a Future that depends on x1 and x2's value. The *Constraint Solving* procedure finds the fixpoint of futures.

The Constraint Collection procedure. The constraint collection algorithm visits program instructions according to the topological ordering of the dominance tree of the source program. Node n_1 dominates node n_2 iff any path from the program root to n_2 goes across n_1. If the source program does not contain uses of undefined variables, then a traversal of its dominance tree in topological order guarantees that every use of a variable v will be seen only after the definition of v [Budimlic et al. 2002]. We traverse the tree, building a partial state according to the type of each node that we visit. The states might contain future values, which will be evaluated in the next phase of our algorithm. However, execution can issue warnings for simple vulnerabilities at this stage. We describe below the construction of states for each type of instruction that we find in Nano-PHP. Figure 6 uses abbreviations for each command; Nano-PHP syntax appears to the right.

An assignment from source, i.e, $v = s, s \in$ SO (source), results in binding v to T. An assignment from a filter $v = f(u)$ results in binding v to C. An assignment from another variable results in binding to the abstract value of that variable. Four possibilities exist for assignment to sink, i.e, $s = v, s \in$ SI (sink). The algorithm ignores the assignment if v is bound to C, it signals a warning of tainting if v is bound to T, a warning of unknown variable if v is bound to ?, or it creates a *delayed check*, if v is bound to a future value. The delayed check is a pair that associates a program instruction with the variable it uses. An alias setup $v_1 = \&v_2$ proceeds as in a concrete semantics. Recall that the e-SSA representation enables the analysis to encode copy of reference with copy of value. The static semantics of a ϕ-function is as follows. If any variable reaching the function is tainted the value of the new definition is tainted, if all reaching variables are clean then the new definition is clean, otherwise the new definition stores a future. Finally, a validation instruction takes as argument the variable in, and two other variables out_1 and out_2, one for each branch. These variables are declared in σ-functions and correspond to versions of in associated with each branch. Variable out_1 defined in the safe branch is clean. Variable out_2 defined in the other branch stores a copy of in's values.

The Constraint Solving procedure. Figure 7 describes the constraint solving procedure for reporting errors due to the access of sink functions to variables storing tainted futures. For that, the procedure looks for variables marked as tainted in a variable dependency chain rooted in variables of delayed checks. For example, the dependency chain $v_1 \rightarrow v_2 \rightarrow \cdots \rightarrow v_n$ indicates a vulnerability iff v_1 appears in a delayed check, v_n is tainted, and there is a path connecting them as defined by the dependency relation from the futures. Very important to note is that only necessary variables are checked.

Complexity. The algorithm from Figure 6 is linear. The time to build the dominance tree of a Nano-PHP program is linear on the number of basic blocks, because each block has at most two successors [Lengauer and Tarjan 1979]. The algorithm from Figure 7 is linear for finding a single warning as all variables visited prior to reaching a tainted or undefined variable must be clean. In the worst case, the algorithm is quadratic on the number of variables if the graph induced by the dependency relation is dense.

```
datatype Abs = C | T | ? | F
// state is a map from variables to abstract values
// all variables initially map to ?
Env: Var -> Abs

// output to next stage.  future checks
DCheck: [Instr x Var]
Futures: Var -> [Var]

// all variables store initially ? in independent locations
foreach instr in topologicalSort(cfg(P))
  switch instr
  case SOURCE(v):   Env := Env \ {(v,T)}     /* v := s, s in SOURCE */
  case FILTER(v):   Env := Env \ {(v,C)}  /* v := f(u), f in FILTER */
  case ASSIGN(v,u): Env := Env \ {(v,Env(u))}             /* v := u */
  case SINK(v):                                 /* s := v, s in SINK */
    switch (Env(v))
      case C: skip                    /* source receives clean value */
      case T: warn-taint(instr)                                /* bug */
      case ?: warn-undef(instr)                     /* possible bug */
      case F: DCheck = (instr, v) : DCheck         /* delayed check */
  case ALIAS(v1,v2): Env := Env \ {(v1,Env(v2))}        /* v1 = &v2 */
  case PHI(out, in_1, ..., in_k):    /* out = phi (in_1, ..., in_k) */
    tmp = {Env(in_i) | 0<i<=k}
    val = if (T in tmp) then T else if (tmp = {C}) then C else F
    Env := Env \ {(out, val)}
    if (val == F) then Futures = Futures \ {(out,[in_1,...,in_k])}
  case GUARD(out_1, out_2, in):                        /* if (in) { */
    Sto := Sto \ {(Env(out_1), C)  /* out_1 = sigma(in)...} else { */
    Sto := Sto \ {(Env(out_2), Env(in))}  /* out_2 = sigma(in)...} */
```

Figure 6. Collect constraints.

```
foreach (instr,var) in DCheck
  isTainted := false; isUndef := false; wlist := [var]; visited := [];
  inner:
  foreach var in wlist
    visited := visited + [var];
    switch (Env(var))
    case C: continue;
    case T: isTainted := true; break inner;
    case ?: isUndef := true;
    case F: wlist := wlist + (Futures(var) - visited);
  if (isTainted)
    warn-taint(instr);
    Env := Env \ {(var,T)}
  else if (isUndef)
    warn-undef(instr);
    Env := Env \ {(var,?)}
  else /* var is clean. reachable state must be clean */
    foreach v in visited
      Env := Env \ {(v,C)}
```

Figure 7. Solve constraints.

4. Experiments

Experimental set up. We have implemented our analysis on top of the phc open source compiler [Biggar et al. 2009], written in C++. We have run our analysis on 42 publicly available PHP applications. We chose benchmarks that have been previously used in the literature [Jovanovic et al. 2006b, Jovanovic et al. 2006b, Xie and Aiken 2006], and that phc can compile. For these experiments, we have configured our tool to use sets of sinks, sources and sanitizers that identify cross-site scripting attacks, which Section 2.1 describes. Notice that, by properly setting these three parameters, our analysis can be easily modified to handle any other kind of vulnerabilities.

Implementation scalability. Out of the 29672 PHP files considered in our evaluation, our tool was able to process 13516. Our tainted flow analysis is not the reason behind this low scalability. The omissions are due to the context sensitive alias analysis used by phc, which consumes too much time and memory, and is unable to handle large PHP files. Because of this limitation, we have set a 3-minutes time limit for phc to finish alias analysis. Nevertheless, the compiler was still able to process a significant fraction of the files and to report non-trivial and previously unknown bugs.

Precision (true and false positives). For this experiment our analysis reported 130 warning messages across 39 distinct PHP files. Manual inspection on each of these warnings revealed actual vulnerabilities in 41 of these reports, i.e., a ~2/3 false positive ratio. We used this list of bugs to perform malicious injections of html code containing JavaScript in 13 distinct PHP files. We have submitted five of the seven bug reports present in Table 2 to bugtraq [7]. The *Sapid* and *phpWebSite* were using an outdated third party library. New versions of this library have fixed these issues.

Table 2 details these numbers for the subjects that contain confirmed vulnerabilities. Column "subject" and "version" give the application name. Column "total" (files) shows the total number of files in the programs, column "processed" shows the total number of these files that phc was able to process, and column "affected" shows the number of files involved in warning reports – for the purpose of inspection it is preferable to have the warnings confined in a small set of files. Average LOCs (commented lines of code) per PHP file are given by columns "total" and "processed". Column "TP: true positive", (respectively, "FP: false-positive") shows the number of confirmed (respectively, spurious) vulnerabilities. False positives arise due to several reasons including: the imprecision of phc's alias-analysis, our inability to process arrays, and the logical infeasibility of some program paths, which static analysis typically fail to identify. Several of our applications use third party software. In particular, four benchmarks out of the seven in Table 2 employed the same spell checking module – a library responsible for six false-positives in each of the four programs. Moreover, 28 out of the 30 bugs reported for `Exponent CMS` were produced by the output of the same tainted variable in different points of one PHP file. These are all different, yet similar bugs.

Efficiency (time $\times$ program size). Figure 8 relates PHP file size with the time to statically analyze it with our tool. Each of the 13,516 data points identifies a run of our tool. We characterize a run with the pair size and time, using a 180-seconds time limit. Note that the analysis of the vast majority of the files, 90.76%, terminates under

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

subject	version	files					warnings	
		total		processed		affected	TP	FP
		#	LOC / #	#	LOC / #			
MODx	1.0.3	472	231	308	228	3	1	2
Exponent CMS	0.97	3456	42	2833	32	3	30	6
DCP Portal	7.0beta	535	97	392	61	7	5	6
Pligg	1.0.4	380	146	179	154	3	1	13
Sapid	r99	359	181	180	202	4	2	7
RunCMS	2.1	737	134	361	86	2	1	6
phpWebSite	1.6.3	1369	213	554	196	2	1	6
avg.	-	-	-	-	-	3.42	5.86	6.57

Table 2. Experimental results of subjects with true alarms.

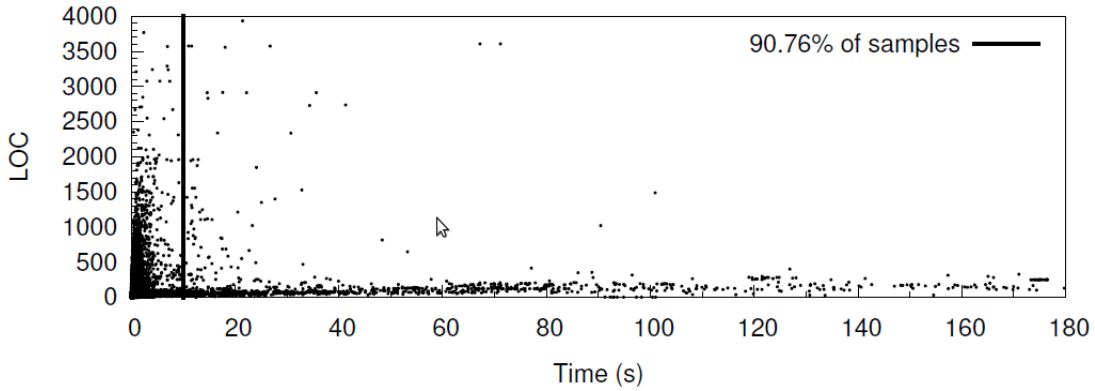


Figure 8. Running time versus program size.

10 seconds. These numbers are on par with numbers from industrial strength analyzers [Jovanovic et al. 2006b], and are orders of magnitude faster than numbers from more precise tools [Wassermann and Su 2007].

5. Related Work

We chose to attack the tainted flow problem via a variation of *data flow analysis*. Data flow analyses are old allies of compiler designers, having been part of compilation systems since the early seventies [Kam and Ullman 1976]. The first influential work to see data flow analysis as a graph reachability problem was introduced by Reps *et al.* [Reps et al. 1995]. The mapping adopted by Reps *et al.* deals with general programs, whereas we use programs in e-SSA form. A disadvantage of the previous approach was the size of the graph that it produces: the number of nodes in the graph is $O(V \times B)$, where V is the number of variables and B is the number of basic blocks in the source program. We avoid this growth, because the e-SSA form tends to increase linearly the number of variables in the source program, and our graph contains $O(V)$ nodes.

Scholz *et al.* [Scholz et al. 2008] have also mapped a flow problem, the *user-input dependence analysis* [Snelting et al. 2006] to an instance of graph-reachability. Scholz's analysis is probably the work that most closely ressembles ours. Scholz *et al.* are interested in finding which program variables might be influenced by input data. Contrary to

our approach, Scholz *et al.* use a program representation called *Augmented Static Single Assignment (a-SSA)* form. The a-SSA form provides information not present in e-SSA form, because it determines which control structures influence program data. That is, in the program `a := read(); c := (a > 0) ? b[0] : 0;` the value of `a` influences the value of `c`, even though these two variables are not related in e-SSA form. However the user-input dependence analysis does not take sanitizers into consideration. Thus, Scholz *et al.*'s a-SSA cannot use information learnt from the outcome of conditionals to bind constraints to variables.

The tainted flow problem is well known in the literature [Jovanovic et al. 2006a, Pistoia et al. 2005, Wassermann and Su 2007, Xie and Aiken 2006]. Wasserman and Su [Wassermann and Su 2007] have used grammars to prove that functions manipulate strings safely. This is an approach very different from ours, as it resorts to a string analysis instead of a data flow analysis. Hence, it is more precise than our approach; yet, much more expensive. Another expensive method, based on symbolic execution, has been described by Xie and Aiken [Xie and Aiken 2006]. The authors, in this case, also assign new symbols to the possible results of conditionals, in a way similar to what e-SSA representation does. However, a direct comparision between their strategy and ours is not possible, because their tool is not publicaly available, and the original paper does not provide enough details to reproduce it. There exist; however, publicaly available tools that perform tainted variable analysis. One of them is MARCO [Pistoia et al. 2005], a Java bytecode analyser. Another is Pixy [Jovanovic et al. 2006a], a PHP analyser. MARCO relies on program slicing [Weiser 1981] to find the set of tainted variables, whereas Pixy uses a monotone framework that associates to each variable, at each program point, a boolean state that defines if the variable is tainted or clean. 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.

6. Conclusion

This paper has presented a new static analysis technique to identify security vulnerabilities related to tainted variable attacks. Our analysis is efficient, because it is equivalent to a graph reachability problem, which we have been able to code as 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 actual bugs in well known web applications. Although we have tested our algorithm in this particular setting, it is general enough to handle tainted variable attacks in different programming languages and in different application domains. As future work, we would like to improve the scalability of phc's alias analysis to handle larger PHP files. The best way to accomplish this is using more efficient context-sensitive techniques, such as PCC's [Bond et al. 2010], for instance.

Acknowledgment: Andrei Rimsa is supported by Capes. We would like to thank Paul Biggar for helping us with phc.

References

Biggar, P., de Vries, E., and Gregg, D. (2009). A practical solution for scripting language compilers. In *SAC*, pages 1916–1923. ACM.

Bodik, R., Gupta, R., and Sarkar, V. (2000). ABCD: eliminating array bounds checks on demand. In *PLDI*, pages 321–333. ACM.

Bond, M. D., Baker, G. Z., and Guyer, S. Z. (2010). Breadcrumbs: efficient context sensitivity for dynamic bug detection analyses. *SIGPLAN Not.*, 45(6):13–24.

Budimlic, Z., Cooper, K. D., Harvey, T. J., Kennedy, K., Oberg, T. S., and Reeves, S. W. (2002). Fast copy coalescing and live-range identification. In *PLDI*, pages 25–32. ACM.

Chow, F. C., Chan, S., Liu, S.-M., Lo, R., and Streich, M. (1996). Effective representation of aliases and indirect memory operations in ssa form. In *CC*, pages 253–267. Springer.

Cytron, R., Ferrante, J., Rosen, B. K., Wegman, M. N., and Zadeck, F. K. (1991). Efficiently computing static single assignment form and the control dependence graph. *TOPLAS*, 13(4):451–490.

Jovanovic, N., Kruegel, C., and Kirda, E. (2006a). Pixy: A static analysis tool for detecting web application vulnerabilities (short paper). In *S&P*, pages 258–263. IEEE.

Jovanovic, N., Kruegel, C., and Kirda, E. (2006b). Precise alias analysis for static detection of web application vulnerabilities. In *PLAS*, pages 27–36. ACM.

Kam, J. B. and Ullman, J. D. (1976). Global data flow analysis and iterative algorithms. *Journal of the ACM*, 23(1):158–171.

Lengauer, T. and Tarjan, R. E. (1979). A fast algorithm for finding dominators in a flowgraph. *ACM Trans. Program. Lang. Syst.*, 1(1):121–141.

Pierce, B. C. (2004). *Types and Programming Languages*. MIT Press, 1st edition.

Pioli, A., Burke, M., and Hind, M. (1999). Conditional pointer aliasing and constant propagation. Technical Report 99-102, SUNY at New Paltz.

Pistoia, M., Flynn, R., Koved, L., and Sreedhar, V. (2005). Interprocedural analysis for privileged code placement and tainted variable detection. In *ECOOP*, pages 362–386.

Reps, T., Horwitz, S., and Sagiv, M. (1995). Precise interprocedural dataflow analysis via graph reachability. In *POPL*, pages 49–61. ACM.

Scholz, B., Zhang, C., and Cifuentes, C. (2008). User-input dependence analysis via graph reachability. Technical report, Sun Microsystems, Inc.

Snelting, G., Robschink, T., and Krinke, J. (2006). Efficient path conditions in dependence graphs for software safety analysis. *TOSEM*, 15(4):410–457.

Stephenson, M., Babb, J., and Amarasinghe, S. (2000). Bidwidth analysis with application to silicon compilation. In *PLDI*, pages 108–120. ACM.

Tavares, A. L. C., Pereira, F. M. Q., Bigonha, M. A. S., and Bigonha, R. (2010). Efficient ssi conversion. In *SBLP*.

Wassermann, G. and Su, Z. (2007). Sound and precise analysis of web applications for injection vulnerabilities. In *PLDI*, pages 32–41. ACM.

Wassermann, G. and Su, Z. (2008). Static detection of cross-site scripting vulnerabilities. In *ICSE*, pages 171–180. ACM.

Weiser, M. (1981). Program slicing. In *ICSE*, pages 439–449. IEEE.

Xie, Y. and Aiken, A. (2006). Static detection of security vulnerabilities in scripting languages. In *USENIX-SS*. USENIX Association.