Verifying Web Applications Using Bounded Model Checking*

Yao-Wen Huang[12], Fang Yu[2], Christian Hang[3], Chung-Hung Tsai[1], D. T. Lee[12], Sy-Yen Kuo[1] †

[1]*Department of Electrical Engineering, National Taiwan University, Taipei 106, Taiwan.* *sykuo@cc.ee.ntu.edu.tw*

[2]*Institute of Information Science, Academia Sinica, Taipei 115, Taiwan* *{ywhuang,yuf,dtlee}@iis.sinica.edu.tw*

[3]*LuFG Informatik II, RWTH Aachen, Ahornstr, 55, 52074 Aachen, Germany* *christian.hang@web.de*

Abstract

The authors describe the use of bounded model checking (BMC) for verifying Web application code. Vulnerable sections of code are patched automatically with runtime guards, allowing both verification and assurance to occur without user intervention. Model checking techniques have relatively complexity compared to the typestate-based polynomial-time algorithm (TS) we adopted in an earlier paper, but they offer three benefits—they provide counterexamples, more precise models, and sound and complete verification. Compared to conventional model checking techniques, BMC offers a more practical approach to verifying programs containing large numbers of variables, but requires fixed program diameters to be complete. Formalizing Web application vulnerabilities as a secure information flow problem with fixed diameter allows for BMC application without drawback. Using BMC-produced counter-examples, errors that result from propagations of the same initial error can be reported as a single group rather than individually. This offers two distinct benefits. First, together with the counterexamples themselves, they allow for more descriptive and precise error reports. Second, it allows for automated patching at locations where errors are initially introduced rather than at locations where the propagated errors cause problems. Results from a TS-BMC comparison test using 230 open-source Web applications showed a 41.0% decrease in runtime instrumentations when BMC was used. In the 38 vulnerable projects identified by TS, BMC classified the TS-reported 980 individual errors into 578 groups, with each group requiring a minimal set of patches for repair.

1. Introduction

As World Wide Web usage expands to cover a greater number of B2B (business-to-business), B2C (business-to-client), healthcare, and e-government services, the reliability and security of Web applications has become an increasingly important concern. A number of deployment-phase mechanisms have recently been examined as a means of assuring the quality attributes of Web applications. One widely-adopted approach to maintaining reliability is applying replication strategies to existing Web applications—that is, enhancing them with fault-tolerance features [28]. To assure security for existing Web applications, Scott and Sharp [24] proposed using gateways that filter invalid and malicious input at the application level. A primary advantage of these two deployment-phase mechanisms is their ability to provide immediate assurance of Web application quality, but their main drawback is that they blindly protect against unpredicted behavior without investigating the actual defects that compromise quality.

To assess Web application quality, Merzbacher and Patterson [18] created a Web application reliability assessment mechanism based on user-experience modeling, and Huang et al. [13] designed a similar security assessment framework that combined user-behavior simulation with user-experience modeling. Both efforts serve as examples of black-box testing. When compared with deployment-phase protection techniques, these approaches emphasize assessment over blind assurance, thus allowing for software improvements. However, they have at least two disadvantages: they do not provide immediate quality assurance, and they cannot guarantee the identification of all flaws. Providing a sound guarantee requires the formalization of Web application bugs and the application of formal verification techniques.

We recently used a combination of static and runtime techniques to create a holistic approach to ensuring Web application quality [14]. The tool that resulted—which we named *WebSSARI* (Web application Security via Static Analysis and Runtime Inspection)—aimed to a) statically verify existing Web application code without any additional annotation effort; and b) after verification, automatically protect potentially defective sections of code. In the project, we formalized Web application vulnerabilities as a secure information flow problem, and based our verification algorithm (TS) on Strom and Yemini's *typestate* [26]—a compile-time technique for

† *Corresponding author: Sy-Yen Kuo. Email: sykuo@cc.ee.ntu.edu.tw Tel: +886-2-2363-5251 ext 444. Fax: +886-2-2367-1909.*

* *This project was supported in part by the National Science Council, Taiwan under grants NSC 92-2213-E-002-011, NSC-93-2422-H-001-0001, and NSC-92-2213-E-001-024.*

verifying program reliability. In many cases, formal verification algorithms for reliability and security verify similar or identical sets of program characteristics. Strom and Yemini's typestate enhances program reliability by detecting (at compile-time) syntactically legal but semantically undefined execution sequences that can lead to unpredictable behavior. Two examples that they give are reading variables before they are initialized and dereferencing pointers after dynamic objects are already deallocated. In information security, the primary objectives are to protect confidentiality, integrity, and availability [22], and null-pointer dereferences are viewed as major causes for denial-of-service vulnerabilities that compromise availability. Furthermore, Strom and Yemini's typestate essentially allows for *flow-sensitive* tracking of program variable states—that is, for each appearance of a variable at a program point, it determines the subset of allowable operations in that specific context. This directly addresses the problem of secure information flow discussed in Section 2.3. In this project, we base our verification on BMC, which allows for precise compile-time estimation of runtime state and thus offers significant improvements over TS. For our experimental tests, we decided to verify security vulnerabilities rather than reliability attributes, although in practice our method can be used for both purposes.

WebSSARI automatically inserts runtime guards in potentially insecure sections of code, meaning that a piece of Web application code will be secured immediately after WebSSARI processing even in the absence of programmer intervention. However, since our initial typestate-based (TS) algorithm sacrificed space and accuracy for speed, it only identified program points that violated safety policies, and was not capable of providing counterexample traces. We identified two major drawbacks from this deficiency:

1. Runtime guards were inserted at program points where safety violations might have occurred (*symptoms*) rather than at points that induced errors (*causes* [2]). Our security policy stated that tainted data could not be used as arguments for calling sensitive functions. Using our TS algorithm meant that runtime guards were inserted at potentially vulnerable function call sites, with the guards sanitizing the tainted variables before they were used as arguments to call sensitive functions. However, following an initial induction, a single piece of tainted data was capable of triggering a snowballing process of propagation and tainting of other data, with the number of tainted variables growing exponentially as the program executed. Without counterexamples, we had to insert a sanitization routine for each instance of variable usage error. A more efficient strategy would be to use an algorithm capable of giving counterexample traces to identify the point where the tainting process begins and to sanitize the data before it propagates.

2. It is very difficult for a programmer to validate a reported error, since any tainting path can potentially spread to numerous function calls. According to our initial experiments, this drawback largely reduces WebSSARI's potential for practical use. Two of the authors needed five days to manually verify all reported vulnerabilities—a labor-intensive task that canceled the benefits resulting from the tool's automated features.

A simple strategy for creating counterexample traces is to adopt model checking techniques, but they are considered very expensive in terms of both time and space complexities. BMC offers a more practical approach to verifying programs containing large numbers of variables, but requires fixed program diameters to be complete [16]. Since we formalize Web application vulnerabilities as a secure information flow problem with fixed diameter, our BMC provides a sound and complete verification mechanism capable of offering counterexamples. Furthermore, since it utilizes ZChaff [19] (an efficient SAT solver that has been used with many industrial projects), it also benefits from ZChaff's many optimization techniques.

In this paper we showed how automated Web application safety verification and assurance can benefit from the counterexample traces, completeness, and soundness associated with a BMC-based approach. Counterexamples and completeness allow for more detailed and informative error reports, thus enhancing the practical potential of WebSSARI. Furthermore, counterexamples allow for more precise identification of locations that require repair, which increases the precision of both the error report generation and the runtime guard instrumentation processes. Completeness and precise instrumentations together result in reduced runtime overhead. Soundness guarantees the absence of bugs.

We previously tested our TS algorithm against 230 real-world Web applications downloaded from SourceForge.net; 69 were identified as having defective code. After notifying the developers, we received 38 acknowledgements and promises of patches. For the present project, we implemented a BMC with ZChaff [19] (a mature SAT solver) and tested it against the same 230 projects (consisting of 1,140,091 statements). The BMC-based approach reduced the number of effective insertions by 41.0% compared to our TS-based results.

2. Web Application Vulnerabilities

As most Web application vulnerabilities arise from the use of untrusted data before sanitization, it is possible to provide an automated patch by inserting sanitization routines at necessary program locations. A major contribution of this paper lies in using counterexample traces to reduce the number of inserted sanitization routines. To make clear how this strategy works, we must first describe how Web application vulnerabilities arise

and how we formalize it. Since we will only provide brief descriptions of the most widely exploited vulnerability—script injection—readers are referred to Scott and Sharp [24], Curphey et al. [10], OWASP (Curphey et al.) [20] [10], and Meier et al. [17] for more details.

2.1. Cross-Site Scripting (XSS)

One severe type of XSS involves the uploading of data by a user, which is then stored for later delivery by a Web application without performing any type of sanitization. Consider the following example: a commercial online auction site hosts a ticket service for users to get support, report bugs, and submit feature requests. Messages posted by users are submitted to a server-side script that inserts them into a backend database. Support tickets and bug reports can only be read by the website's support personnel, but feature requests can be read by all users. When viewing tickets, a request is sent to a server-side script that retrieves corresponding data from the backend database and constructs HTML output. If a user submits a bug report (or a feature request) that contains a piece of malicious script, the script will be delivered to the support personal (or other users) on behalf of the Web server. This grants rights that the script normally would not receive. Figure 1 presents a simplified version of a vulnerability that our WebSSARI discovered in *PHP Support Tickets*.

```
$query="INSERT INTO tickets_tickets(tickets_id,
tickets_username,tickets_subject, tickets_question)
VALUES("$_SESSION['username']"", ""$_POST['ticketsubject']",
"$_POST['message']")";
$result = @mysql_query($query);
```

Figure 1. A XSS vulnerability found in *PHP Support Tickets* code for ticket submission.

Note that user input values "ticketsubject" and "message" have been inserted into the database without sanitization. An example of code from *PHP Support Tickets* that uses the backend database to generate HTML output for displaying tickets is shown in Figure 2. Since the value "tickets_subject" (containing untrusted data submitted by the user) is used without sanitization to construct HTML output, the code is vulnerable to XSS.

```
$query="SELECT tickets_id, tickets_username,
tickets_subject FROM tickets_tickets";
$result = @mysql_query($query);
WHILE ($row = @mysql_fetch_array($result)) {
  extract($row);
  echo"$tickets_username<BR>$tickets_subject<BR><BR>"
}
```

Figure 2. Simplified code for displaying the tickets.

2.2. SQL Injection

Considered more severe than XSS, SQL injection vulnerabilities occur when untrusted values are used to construct SQL commands, resulting in the execution of arbitrary SQL commands given by an attacker. The example we offer below is based on a vulnerability we discovered in *ILIAS Open Source*, a popular Web-based learning management system.

```
$sql="INSERT INTO track_temp VALUES('$HTTP_REFERER');"
mysql_query($sql);
```

Figure 3. A simplified SQL injection vulnerability found in *ILIAS Open Source*.

In Figure 3, $HTTP_REFERER (a global variable set by the Web server to indicate the referrer field of a HTTP request) is used to construct a SQL command. The referrer field of a HTTP request is an untrusted value given by the HTTP client; an attacker can set the field to:

```
');DROP TABLE ('users
```

This will cause the code in Figure 3 to construct the $sql variable as:

```
INSERT INTO track_temp VALUES('');
DROP TABLE ('users');
```

Table "users" will be dropped when this SQL command is executed. This technique, which allows for the arbitrary manipulation of backend database, is responsible for the majority of successful Web application attacks. During our experimentation with WebSSARI, we found that developers who acknowledge that variables from HTTP requests should not be trusted tend to forget that the same holds true for the referrer field, user cookies, and other types of information collected from HTTP requests.

2.3. Specifying Web Application Reliability and Security Policies

According to our examples, compromises in integrity lead to compromises in confidentiality and availability. When untrusted data is used to construct trusted output without sanitization, violations in data integrity occur, leading to escalations in access rights that result in availability and confidentiality compromises. There is a clear need for a mechanism that specifies and enforces legal information flow policies within Web application programs. This can be achieved by assigning a "state" that represents a variable's current trust level. The challenge is to design a compile-time algorithm that predicts variable runtime states at each program point—similar to the requirement for enforcing certain reliability policies. Strom and Yemini's [26] typestate algorithm for enhancing software reliability (i.e., checking for un-initialized variables and illegal pointer dereferences) is one example of an algorithm that offers compile-time tracking of variable states.

3. Verification algorithm

Most Web applications are written in script languages, including PHP, ASP, Perl, and Python [15]. We designed our verification algorithm to take advantage of the imperative, deterministic, and sequential characteristics of these programming languages. In our information flow model, we associate every program variable x with a safety type t_x, which represents the current safety level of x. To verify a program, we first generate an abstract interpretation (AI) of a program [9] that retains the program's information flow properties. We then use bounded model checking to verify the correctness of all possible safety states of the AI.

3.1. Information Flow Model

To allow for arithmetic on variable safety types, we followed Denning's [12] model and made the following assumptions:

1. Each variable is associated with a safety type.
2. T = $\{\tau_1, \tau_2, \cdots, \tau_n\}$ τ is a finite set of safety types.
3. T is partially ordered by $\leq$, which is reflexive, transitive, and antisymmetric. For $\tau_1, \tau_2 \in T$,

   $\tau_1 = \tau_2$ iff $\tau_1 \leq \tau_2$ and $\tau_2 \leq \tau_1$,

   and $\tau_1 < \tau_2$ iff $\tau_1 \leq \tau_2$ and $\tau_1 \neq \tau_2$.
4. (T, $\leq$) forms a complete lattice with a lower bound $\perp$ such that $\forall \tau \in \mathrm{T}, \perp \leq \tau$, and an upper bound $\top$ such that $\forall \tau \in \mathrm{T}, \tau \leq \top$.

These assumptions imply that a greatest lower bound operator and a least upper bound operator exit on T. For subset Y $\subseteq$ T, let $\sqcap$Y denote $\top$ if Y is empty and the greatest lower bound of the types in Y otherwise; let $\sqcup$Y denote $\perp$ if Y is empty and the least upper bound of the types in Y otherwise.

Types resulting from expressions are determined using the upper- and lower-bound operators (i.e., $\sqcup$ and $\sqcap$, respectively) defined above.

3.2. Abstract Interpretation

Given a program p, we first use a filter to generate *F(p)*, which consists of command sequences constructed according to the following syntax rules:

(commands)

$c ::= x := e \mid f_i(X) \mid f_o(X) \mid stop \mid$ if e then c_1 else $c_2 \mid$ while e do $c \mid c_1; c_2$

(expression)

$e ::= x \mid n \mid e_1 \sim e_2$

, where x is a variable, n is a constant, ~ stands for binary operations such as +, and $X \subseteq \mathrm{dom}(p)$ is a variable set. By preserving only assignments, function calls and conditional structures. *F(p)* unfolds function calls and discards all program constructs that are not associated with information flow. During execution, data is retrieved from external sources (e.g., reading files or retrieving environment variables). Functions that permit the flow of external data into a program are identified as *untrusted input channels* (UIC), denoted as *fi(X)*. An example in PHP is GET_HTTP_VARS(), which retrieves data from HTTP requests sent by Web clients. In WebSSARI, UICs are given predefined *postconditions* consisting of command sets that match the designated safety levels of the retrieved data. At the same time, program execution also entails calling functions that manipulate system resources or affect system integrity—in PHP, for instance, exec() executes system commands and echo() generates output. These functions—identified as *sensitive output channels* (SOC) and denoted as *fo(X)*—require trusted arguments. Each one is assigned a predefined *precondition* that states the required argument safety levels.

Filtered Result: F(p)	Abstract Interpretation: AI(F(p))
x = e;	$t_x = t_e$, where $t_n = \perp$, $t_{e_i \sim e_j} = t_{e_i} \sqcap t_{e_j}$
fi(X);	$\forall_{x \in X} t_x = \tau$, where τ depends on the postconditions of *fi*
fo(X);	$assert(X, \tau_r)$, where τ_r depends on the preconditions of *fo*
stop;	stop;
if *e* then c_1 else c_2	if b_e then *AI(c_1)* else *AI(c_2)*
while *e* do *c*	if b_e then *AI(c)*
c_1; c_2	AI(c_1); AI(c_2)

Figure 4. Abstract interpretation procedure.

As one would expect, the *stop* command immediately terminates program execution. When verifying conditional structures within a program, we do not consider how condition *e* evaluates, but rather focus on making sure that each branched path behaves correctly. In path-sensitive terminology, we treat all these conditions as nondeterministic conditions. Furthermore, since we only consider information flow, loop structures can be deconstructed into selection structures.

Consequently, given a safety type lattice T and sets of pre- and postconditions, we translate *F(p)* into an AI that consists of only if instructions, type assignments and assertions. The intuitive interpretation procedure *AI(F(p))* is illustrated in Figure 4. An assignment from expression *e* to variable *x* is translated into a *type assignment* that conveys *e*'s safety type to *x*. Function preconditions are expressed using assertions ($assert(X, \tau_r)$) that specify safety requirements ($\forall_{x \in X}, t_x < \tau_r$, $\tau_r \in \mathrm{dom}(T)$), meaning that the types of all variable in X must be lower (safer) than the τ_r. Postconditions are expressed using

type assignment sets (in the form $\forall_{x \in X} t_x = \tau$, $\tau \in \mathrm{dom}(T)$) that describe the safety level of each piece of retrieved data. In WebSSARI, pre- and postcondition definitions are stored in two prelude files that are loaded during startup [14].

3.3. Formal Verification

Using the AI definitions given above, the verification of a given PHP program *p* consists of checking whether *AI(F(p))* is consistent with its assertions. We will present some observations before attempting to automate this process. First, $AI(F(p))$ is loop-free and its flow chart forms a directed acyclic graph (DAG), implying a fixed program diameter [16]. Second, *AI(F(p))* represents a single sequential process with large numbers of: a) variables and b) branches following simple assignments. Based on these observations, we adopted Bounded Model Checking [3] [4] rather than the more conventional model checking techniques based on BDDs (binary decision diagrams). In addition to its ability to provide both sound and complete verification. BMC is more efficient in finding bugs and more capable of handling large numbers of variables that may cause BDDs to crash [23].

In BMC, a system's transition relations are unfolded with bounded steps and coupled with initial and risk conditions to form a CNF formula, which is then solved using a SAT solver. In the following sections we will describe the details of our program encoding and some of the difficulties we encountered.

3.3.1. Encoding using an auxiliary variable

The first challenge was keeping track of program behaviors. A naïve but conceptually straightforward solution was to add an auxiliary variable *l* to record program lines (statements). Given a program *p*, let $X = l \cup \{t_x \mid x \in \mathrm{dom}(AI(F(p)))\}$ denote state variables, then we can construct a control-flow graph *CFG(X,p)*. The transition relations of *CFG(X,p)* are encoded as a CNF formula, *T(s,s')*, where *s* and *s'* denote the current and subsequent program states (the evaluation of X), respectively. By rolling $T(s^i, s^{i+1})$ up for a bounded k number of steps (the length of the longest path in *CFG(X,p)*), the entire formula is represented as:

$$B(X,k) = I(s^0) \wedge T(s^0, s^1) \wedge \cdots \wedge T(s^{k-1}, s^k) \wedge R(s^i, \cdots, s^k),$$

where $I(s^0)$ is the initial condition and $R(s^i, \cdots, s^k)$ specifies the risk conditions (assertion negations) within the *i*th and *k*th states.

We incorporated this idea into our first BMC version, xBMC0.1, but initial experiments revealed frequent system breakdowns, primarily due to inefficiently encoding each assignment using 2|*X*| variables.

3.3.2. Encoding using variable renaming

Clarke et al.[6] [7] automated memory overflow and assertion consistency tests for C and Verilog with their CBMC [8] tool. The checker unwinds C or Verilog programs and converts them to a Boolean formula that can be checked for behavior consistency. CBMC uses variable renaming to create a *single assignment* program—similar to a *Static Single Assignment* (SSA) program [11] without the $\phi-$ condition. Since the algorithm uses variable renaming, it encodes each assignment using only 2 variables. However, compared with the algorithm mentioned in the previous section, using renaming makes it inefficient for modeling languages containing loop constructs. Since our information flow model allows for deconstructing loop structures into selection structures without unfolding, we can adopt Clarke et al.'s algorithm without any drawback.

In their algorithm, AI variables are renamed such that each renamed variable is assigned only once. Assume that variable *v* is referred to at program location *i* in an AI. Let α denote the number of assignments made to variable *v* prior to *i*, then the variable *v* at location *i* is renamed to. v_α. After this renaming procedure ρ, each assignment becomes unique, which allows for information flow tracking minus the need for auxiliary location variables. In our revised version (xBMC1.0), we used Clarke et al.'s [6] [7] algorithm to encode our AI. Given a command *c*, the constraint generated by procedure C(*c*,*g*) is shown in Figure 5 (*g* denotes the guard and is initially true).

AI Command	Constraint
stop; or empty	$C(c,g) \coloneqq true$
$t_x = t_e$	$C(c,g) \coloneqq {t^i}_x = g \,?\, \rho(t_e) : {t^{i-1}}_x$
$assert(\sqcap(t_x \mid x \in X) < T_R)$	$C(c,g) \coloneqq g \Rightarrow \rho\left(\bigcap_{x \in X} t_x < T_R\right)$
if b_e then c_1 else c_2	$C(c,g) \coloneqq C(c_1, g \wedge b_e) \wedge C(c_2, g \wedge \neg b_e)$
$c_1; c_2$	$C(c,g) \coloneqq C(c_1,g) \wedge C(c_2,g)$

Figure 5. Constraint construction procedure.

However, instead of checking all assertions at the same time [6] [7], we check one assertion at a time and generate all counterexamples for that assertion. For each assertion, we generate a formula *B* and check for its satisfiability. If *B* is satisfiable, we obtain a counterexample and make *B* more restrictive by negating out the counterexample. We iterate this loop until *B* is unsatisfiable—that is, when we have collected all counterexamples. When generating a formula for an assertion $assert_i$, we view it and all its preceding commands as a concatenation $c;assert_i$. The corresponding formula B_i is then constructed with the

PHP source code	Filtered Result	Abstract Interpretation	Renaming	Constraints for each assertion

…
if (Nick) {
  $tmp=$_GET["nick"];
  echo
  (htmlspecialchars
  ($tmp)
  );
} else {
  $tmp="You are the".
  $GuestCount." guest";
  echo($tmp);
}
…
	…
	if (Nick) {
	  $f^t{}_i(nick)$;
	  $tmp = nick$;
	  $f^u{}_i(tmp)$;
	  $f_o(tmp)$;
	} else {
	  $tmp = GuestCount$;
	  $f_o(tmp)$;
	}
	…
		…
		if b_{Nick} then
		  $t_{nick} = T$;
		  $t_{tmp} = t_{nick}$;
		  $t_{tmp} = U$;
		  assert$(t_{tmp} < T)$;
		else
		  $t_{tmp} = t_{GuestCount}$;
		  assert$(t_{tmp} < T)$;
		…
			…
			If b_{Nick} then
			  $t^i{}_{nick} = T$;
			  $t^j{}_{tmp} = t^i{}_{nick}$;
			  $t^{j+1}{}_{tmp} = U$;
			  assert$_k(t^{j+1}{}_{tmp} < T)$;
			else
			  $t^{j+2}{}_{tmp} = t^k{}_{GuestCount}$;
			  assert$_{k+1}(t^{j+2}{}_{tmp} < T)$;
			…
				$B_k := (t^i{}_{nick} = b_{Nick} ? T : t^{i-1}{}_{nick}) \wedge$
				$(t^j{}_{tmp} = b_{Nick} ? t^i{}_{nick} : t^{j-1}{}_{tmp}) \wedge$
				$(t^{j+1}{}_{tmp} = b_{Nick} ? U : t^j{}_{tmp}) \wedge$
				$\neg(b_{Nick} \Rightarrow t^{j+1}{}_{tmp} < T)$
				$B_{k+1} := (t^i{}_{nick} = b_{Nick} ? T : t^{i-1}{}_{nick}) \wedge$
				$(t^j{}_{tmp} = b_{Nick} ? t^i{}_{nick} : t^{j-1}{}_{tmp}) \wedge$
				$(t^{j+1}{}_{tmp} = b_{Nick} ? U : t^j{}_{tmp}) \wedge$
				$(b_{Nick} \Rightarrow t^{j+1}{}_{tmp} < T) \wedge$
				$(t^{j+2}{}_{tmp} = \neg b_{Nick} ? t^k{}_{GuestCount} : t^{j+1}{}_{tmp}) \wedge$
				$\neg(\neg b_{Nick} \Rightarrow t^{j+2}{}_{tmp} < T)$

Figure 6. An example of translation from PHP to Boolean formulas.

negation of $assert_i$: $B_i := C(c, g) \wedge \neg C(assert_i, g)$. Examples are given in Figure 6.

CNF(B_i) transforms B_i into a CNF formula, which can then be solved using the efficient SAT solver zChaff [19]. As mentioned above, if CNF(B_i) is satisfiable, zChaff proposes a truth assignment α_i that violates $assert_i$. Let $\mathbf{B_N}$ denote the set of all nondeterministic boolean variables in the AI. According to $\mathbf{B_N}$'s values in α_i, we can trace the AI and generate a sequence of single assignments, which represents one counterexample trace. In order to collect all possible counterexamples, we iteratively make B_i more restrictive until it becomes unsatisfiable. In other words, each time a truth assignment ${\alpha_i}^j$ is proposed at the jth iteration, we generate the negation clause ${N_i}^j$ of $\mathbf{B_N}$. Thus the more restrictive formula ${B_i}^j$ at the $j+1$th iteration is defined as ${B_i}^j = B_i \wedge N_i^1 \wedge \cdots \wedge N_i^j$.

Once the formula becomes unsatisfiable, we continue the constraint generation procedure (see Figure 5) $C(c, g) := C(c_1, g) \wedge C(assert_i, g)$ until we encounter the next assertion. Since the AI program is loop-free, each assertion will be checked exactly once.

3.3.3. Counterexample Analysis

For any instance of unsafe code reported by BMC, WebSSARI inserts guards that perform run-time inspections as part of an automated patching process—specifically, it inserts routines that sanitize untrusted input before they are used by SOCs. Several combinations of patching locations may fix the same section of insecure code. In this section, we will describe how our algorithm takes advantage of BMC-produced counterexamples to identify an optimal patching combination with a minimum number of insertions. Our definition of an effective fix is as follows.

Definition 1: Given a error trace r, a Fix(V) is said to effective if, after sanitizing all variables $v \in V$, the error trace is removed (fixed).

For an error trace r, we refer to the set of variables that *directly* caused assertion violations (i.e., variables that appeared in assertion commands and caused violations) as *violating variables*, and a variable set that yields an effective fix when sanitized as a *fixing set*. Given an error trace set R, our goal is to find a minimal fixing set V_R such that for every trace $r \in R$, Fix(V_R) serves as an effective fix of r. For an error trace $r \in R$, let V_r denote the violating variables of r. A naïve method of finding a fixing set for R is $V^n{}_R = \bigcup_{\forall r \in R} V_r$. Obviously, fixing all violating variables in R removes all error traces (for all $r \in R$, Fix($V^n{}_R$) is an effective fix), but in many cases, $V^n{}_R$ is not the minimum set. Figure 7 presents a simplified version of a vulnerable file we found in *PHP Surveyor*. In this example, the tainted variable $sid taints $iquery, $i2query, and $fnquery, causing lines 2, 3, and 4 to become vulnerable. A naïve fixing set would be {$iquery, $i2query, $fnquery}—as was adopted by our TS algorithm. However, the optimal fixing set is clearly {$sid}, and so sanitizing $sid is by itself an effective fix. In a source code of *PHP Surveyor*, $sid was the root cause of 16 vulnerable program locations; our TS algorithm made 16 instrumentations, whereas a single instrumentation would have been sufficient to secure the code.

```
1: $sid = $_GET['sid']; if (!$sid) {$sid = $_POST['sid'];}
2: $iq = "SELECT * FROM groups WHERE sid=$sid";DoSQL($iq);
3: $i2q = "SELECT * FROM ans WHERE sid=$sid; DoSQL($i2q);
4: $fnquery = "SELECT * FROM questions, surveys WHERE
   questions.sid=surveys.sid AND questions.sid='$sid'";
   DoSQL($fnquery);
```

Figure 7. Multiple vulnerabilities arising from the same root cause in PHP Surveyor.

To achieve this, for each violating variable $v_\alpha \in V_r$, a replacement set s_{v_α} is built by tracing back from the violation point along the error trace r while recursively

adding variables that serve as unique r-values of single assignments. That is,

$$s_{v_\alpha} = \begin{cases} \{v_\alpha\} \cup s_{v_\beta}, & \text{if the single assignment is} \\ & \text{in the form } v_\alpha = v_\beta \\ \{v_\alpha\}, & \text{otherwise.} \end{cases}$$

Note that if v_α is tainted, then s_{v_α} presents a tainted flow path along which subsequent assignments cause v_α to become tainted. While tracing back along the error trace r, s_{v_α} is expanded with variables that can be sanitized instead of v_α, yet achieve the same effect as sanitizing v_α.

Lemma 1: If a Fix(V_r), $v_\alpha \in V_r$, is an effective fix for r, then for any $v_\beta \in s_{v_\alpha}$, Fix($V_r[v_\beta / v_\alpha]$) is also an effective fix, where $V_r[v_\beta / v_\alpha]$ denotes $(V_r - \{v_\alpha\}) \cup \{v_\beta\}$.

Proof: Initially, s_{v_α} is expanded only with single assignments in the form v_α=v_β, meaning that the value of v_α is solely dependent upon v_β. After expanding s_{v_α} with v_β, this process is repeated to add variables (if any) whose value v_β depends on. Therefore, sanitizing any variable in s_{v_α} has the same effect as sanitizing the initial variable v_α.

To identify the root errors, we calculate an error trace set R's minimum fixing set. First, for each $r \in R$, we identify the violating variable set V_r and then apply the naïve method to derive $V^n{}_R$, where $V^n{}_R = \bigcup_{\forall r \in R} V_r$.

Second, for each $v_\alpha \in V^n{}_R$, we calculate its replacement set s_{v_α}. Finally, the minimum effective fixing set $V^m{}_R$ can be obtained by solving $\min |V^m{}_R|$ such that $\forall_{v_\alpha \in V^n{}_R} s_{v_\alpha} \cap V^m{}_R \neq \varnothing$.

Lemma 2: Given an error trace set R, for all $r \in R$, Fix($V^m{}_R$) is an effective fix for r.

Proof: Obviously, fixing all violating variables in R removes all error traces. That is, if $V^n{}_R = \bigcup_{\forall r \in R} V_r$, then for all $r \in R$, Fix($V^n{}_R$) is an effective fix. From Lemma 1, substituting $v_\alpha \in V_r$ with variable $v \in s_{v_\alpha}$ still yields an effective fix. Since for all $v_\alpha \in V^n{}_R$, for all s_{v_α} of R, $s_{v_\alpha} \cap V^m{}_R \neq \varnothing$, Fix($V^m{}_R$) remains an effective fix for all $r \in R$.

3.3.4. Minimal Fixing Set

The only remaining problem is solving $\min |V^m{}_R|$ such that $\forall_{v_\alpha \in V^n{}_R} s_{v_\alpha} \cap V^m{}_R \neq \varnothing$. In the following, we offer a formal definition and prove it to be a NP-complete problem. Finally, we describe the heuristic search procedure that WebSSARI incorporates.

Definition 2*: Given a variable set V and a collection of subsets of V, $S = \{S_1, \cdots, S_n\}$, the MINIMUM-INTERSECTING-SET problem is to find a minimum set $M \subseteq V$, such that for $1 \le i \le n, S_i \cap M \neq \varnothing$.*

Theorem: *The MINIMUM-INTERSECTING-SET problem is NP-complete.*

Proof: The MINIMUM-INTERSECTING-SET (MIS) problem is in NP since we can guess a subset *S'* of *S* and check in polynomial time whether a) *S'* shares at least one common element with every set in $Si \in S$, and b) *S'* has an appropriate size. To prove that it is NP-complete, we reduce the VERTEX-COVER problem to MIS. If G is an undirected graph, a vertex cover of G is a subset of the nodes where every edge of G touches one of those nodes. The vertex cover problem asks for the size of the smallest vertex cover. Our reduction maps each edge e_i in G to a set $S_i = \{v, v'\}$, where v and v' denote the two vertices connected by e_i. Then the undirected graph G with edges $e_1, \cdots, e_n$ can be represented as a set of vertex sets $S = \{S_1, \cdots, S_n\}$. The size of the smallest vertex cover is $|M|$ where $\forall Si \in S$, $Si \cap M \neq \varnothing$.

The MIS problem can be reduced to the SET-COVER problem where all sets have an equal cost. Therefore, in WebSSARI, we adopted the greedy heuristic algorithm in [5], which gives a 1+ln(|*S*|) approximation ratio in polynomial time. The reduction procedure is described as follows. Given a universe *U* of *n* elements, a collection of subsets of *U*, $S_V = \{S_{v_1}, \cdots, S_{v_k}\}$, and a cost function $c : S_V \to Q^+$, the SET-COVER problem asks to find a minimum-cost subcollection of S_v that covers all elements of *U*. The reduction takes each S_i as an element of *U*. Let $S_V = \left\{ S_{v_i} \mid v_i \in V \right\}$, where $S_{v_i} = \{S_j \mid v_i \in S_j, S_j \in U\}$, and $c\left(S_{v_i}\right)$ be a constant. The MIS problem can be solved by first solving this SET-COVER problem and then replacing each selected S_{v_i} with v_i.

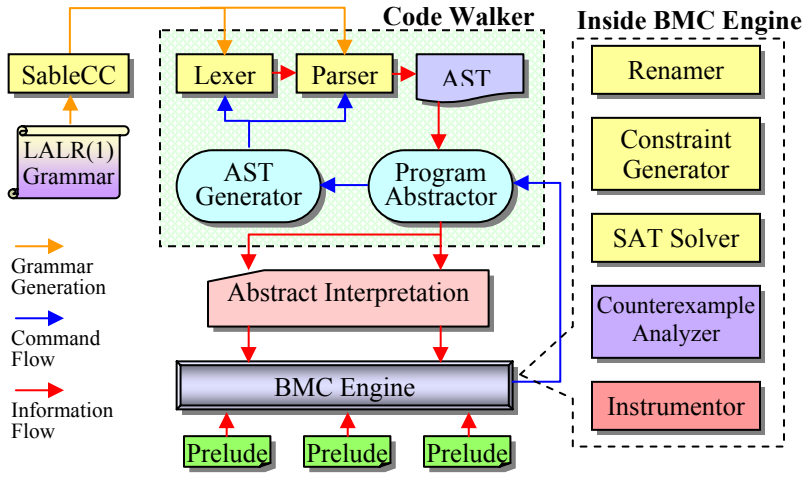

Figure 8. WebSSARI system architecture.

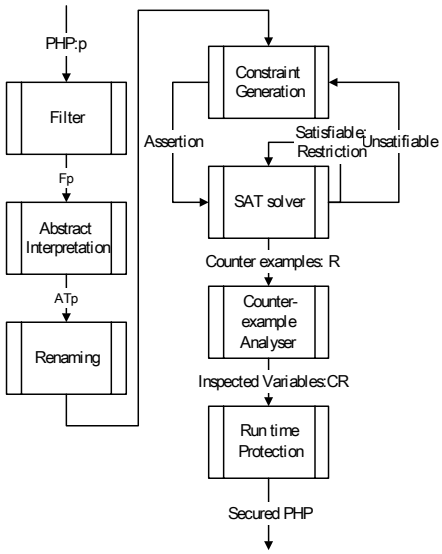

Figure 9. The verification process.

4. System implementation

To test our approach, we developed WebSSARI to verify real-world Web applications. An illustration of WebSSARI's system architecture is presented in Figure 8. A *code walker* consists of a lexer, a parser, an AST (abstract syntax tree) maker, and a *program abstractor*. The program abstractor asks the AST maker to generate a full representation of a program's AST. The AST maker uses the lexer and the parser to perform this task, handling external file inclusions along the way. By traversing the AST, the program abstractor generates an AI. Using the algorithms described in Section 3, the *BMC engine* performs verification of the AI. For each variable involved in an insecure statement, it inserts a statement that secures the variable by treating it with a sanitization routine. Sanitization routines are stored in a prelude, and users can supply the prelude with their own routines. The whole AI verification process is illustrated in Figure 9.

5. Experimental results

SourceForge.net [1], the world's largest open-source development website, classifies all projects according to language, purpose, popularity, and development status (also referred to as maturity). As part of our previous TS algorithm effort, we established a sample of 230 projects (written in PHP) reflecting a broad variation in terms of the SourceForge.net classifications. After downloading their sources and testing them with WebSSARI, we manually inspected every single report of a security violation. If we identified an actual vulnerability, we sent an email notification to the developer. Of the 69 developers we contacted, 38 acknowledged our findings and stated that they would provide patches (Figure 10).

Project	A	TS	BM	Project	A	TS	BMC
GBook MX	60	4	2	SquirrelMail	99	7	7
AthenaRMS	0	3	2	PHPMyList	69	10	4
PHPCodeCabinet	71	25	25	EGroupWare	99	4	4
BolinOS	94	3	3	PHPFriendlyAdmin	87	16	16
PHP Surveyor	99	169	90	PHP Helpdesk	87	1	1
Booby	90	5	4	Media Mate	0	53	16
ByteHoard	98	2	2	Obelus Helpdesk	22	8	6
PHPRecipeBook	99	11	8	eDreamers	80	7	1
phpLDAPadmin	97	25	13	Mad.Thought	66	4	4
Segue CMS	77	11	9	PHPLetter	79	23	23
Moregroupware	99	7	7	WebArchive	2	7	2
iNuke	0	3	3	Nalanda	58	27	8
InfoCentral	82	206	57	Site@School	94	46	40
WebMovieDB	24	7	5	PHPList	0	16	1
TestLink	88	69	48	PHPPgAdmin	98	3	3
Crafty Syntax Live Help	96	16	1	Anonymous Mailer	73	7	7
ILIAS open source	20	2	2	PHP Support Tickets	0	40	40
PHP Multiple Newsletters	68	30	30	Norfolk Household Financial Manager	0	60	60
International Suspect Vigilance Nexus	0	20	12	Tiki CMS Groupware	99	12	12
				Total		980	578

A: Project activity
TS: TS-reported errors BMC: BMC-reported errors

Figure 10. The number TS- and BMC-reported errors of the 38 projects that responded to our notifications.

The 230 projects consisted of 11,848 files consisting of 1,140,091 statements; 515 files were identified by TS as vulnerable. Soon after starting the task of manually validating all reported vulnerabilities, the authors realized that the lack of counterexamples made for a laborious and

time-consuming task that required investigating multiple function calls spanning multiple files. In an effort to speed up the process, we added features to the WebSSARI GUI that helped users: a) navigate between different source files, function calls, and vulnerable lines; b) identify particular variables (such as highlighting variables that caused assertions); and c) search for specific variables or text patterns. However, even with these special features, the job of manual validation remained difficult. We therefore added a tool called PHPXREF [27] to generate cross-referenced HTML documentations of source code. Despite the enhancements, it still took two of the authors four full working days to validate the 515 files that were identified as vulnerable.

In the revised project that is the focus of this paper, we used BMC to provide counterexample traces. Differences in the TS and BMC reports on the 38 vulnerable projects whose developers acknowledged our findings are shown in Figure 10. For these projects, the total number of vulnerable statements originally reported by TS was 980. Using the same test set, BMC reported a total of 578 error introductions, meaning that the 980 vulnerabilities were caused by the propagation of 578 errors. Compared with TS, this process yielded an additional 41.0 percent reduction in the number of instrumentations.

6. Discussion

In this project, we used BMC-provided counterexamples to identify the cause of errors, which increases the precision of both error reports and code instrumentation. In a very recent project, Ball, Naik, and Rajamani [2] made a very similar effort—they attempted to enhance their model checker SLAM with the ability to *localize* errors. As they mentioned, current model checkers report error *symptoms* rather than the actual *causes*. Furthermore, even the state-of-the-art model checkers today report only a single error trace per run. They reported their experiences in using their algorithm to detect locking bugs in C device drivers.

Our efforts were motivated by our previous effort in verifying Web applications using the TS algorithm. TS reported individual error symptoms, which not only resulted in inefficient automated patching, but also made it difficult to report a meaningful number of discovered vulnerabilities, since many of the reported errors were attributed to a same cause and should not have been double counted. Ball, Naik, and Rajamani focused on locking bugs, which usually have a one-to-one mapping between a symptom and a cause. However, we focus on information flow bugs, which are much more complex and can have a many-to-many symptom-cause mapping—the reason why localizing errors resulted in a *MINIMUM-INTERSECTING-SET* problem. Furthermore, their efforts mainly contributes to more informative error reports, while ours also results in more efficient automated patching. Like Ball, Naik, and Rajamani's algorithm, ours also requires that all counterexample traces be identified. However, since SLAM is a BDD-based model checker and xBMC is a SAT-based bounded model checker, our proposed method for extracting all counterexamples is unique from theirs.

7. Conclusion

In this paper we proposed a practical approach for formally verifying Web application reliability and security. In an earlier work, we used a typestate-based algorithm (TS) that essentially performs breadth-first searches on control flow graphs and trades space for time. Although it has polynomial-time complexity, it is incapable of providing counterexample traces. This proved to be a major deficiency that reduced WebSSARI's potential for practical use. On the other hand, we considered a depth-first search algorithm to be too costly in terms of time, and so we implemented a bounded model checker using ZChaff [19] (a mature SAT solver) and used it to produce counterexample traces.
Two immediate benefits of counterexample traces are a) they allow for more informative error reports, and b) they can be used to identify multiple errors (symptoms) with the same root cause. Such information not only contributes to greater report accuracy, but also sharply reduces the number of inserted runtime guards. We showed that the problem of finding the minimum error causes (groups) is NP-complete, and offered a greedy heuristic-based strategy.

8. Acknowledgement

We deeply appreciate the anonymous reviewers for offering us many valuable comments. We would also like to thank Dr. Bow-Yaw Wang for his useful suggestions.

9. References

[1] Augustin, L., Bressler, D., Smith, G. "Accelerating Software Development through Collaboration." In *Proc. 24th International Conf. Software Engineering*, p.559-563, Orlando, Florida, 2002.

[2] Ball, T., Naik, M., Rajamani, S. "From Symptom to Cause: Localizing Errors in Counterexample Traces." In *Proc. 30th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, p.97-105, New Orleans, Louisiana, 2003.

[3] Biere, A., Cimatti, A., Clarke, E. M., Fujita, M., Zhu, Y. "Symbolic Model Checking without BDDs." In *Proc. 5th Int'l Conf. Tools and Algorithms for Construction and Analysis of Systems*, p.193-207,

volume LNCS 1579, Amsterdam, The Netherlands, 1999. Springer-Verlag.

[4] Biere, A., Cimatti, A., Clarke, E. M., Fujita, M., Zhu, Y. "Symbolic Model Checking using SAT Procedures instead of BDDs." In *Proc. 36th Design Automation Conference*, p.317-320, New Orleans, Las Angeles, 1999.

[5] Chvatal, V. "A Greedy Heuristic for the Set Covering Problem." Mathematics of Operations Research, 4:33-235, 1979.

[6] Clarke, E., Kroening, D., Yorav, k. "Behavioral Consistency of C and Verilog Programs using Bounded Model Checking." Technical Report CMU-CS-03-126, Carnegie Mellon University, School of Computer Science, 2003.

[7] Clarke, E., Kroening, D., Yorav, K. "Behavioral Consistency of C and Verilog Programs using Bounded Model Checking." In *Proc. 40th Design Automation Conference*, Session 23.3, Anaheim, CA, 2003.

[8] Clarke, E., Kroening, D. "ANSI-C Bounded Model Checker User Manual." Carnegie Mellon University, School of Computer Science, 2003.

[9] Cousot, P., Cousot, R. "Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Constructions or Approximation of Fixpoints." In *Conference Record of the 4th ACM Symp. Principles of Programming Languages*, p.238-252, 1977.

[10] Curphey, M., Endler, D., Hau, W., Taylor, S., Smith, T., Russell, A., McKenna, G., Parke, R., McLaughlin, K., Tranter, N., Klien, A., Groves, D., By-Gad, I., Huseby, S., Eizner, M., McNamara, R. "A Guide to Building Secure Web Applications." The Open Web Application Security Project, v.1.1.1, Sep 2002.

[11] Cytron, R., Ferrante, J., Rosen, B. K., Wegman, M. N., Zadeck, F. K. "An Efficient Method of Computing Static Single Assignment Form." In *Proc. 16th ACM SIGPLAN-SIGACT Symp. Principles of Programming Languages*, p.25-35, Austin, Texas, 1989. ACM Press.

[12] Denning, D. E. "A Lattice Model of Secure Information Flow." Communications of the ACM, 19(5):236-243, 1976.

[13] Huang, Y. W., Huang, S. K., Lin, T. P., Tsai, C. H. "Web Application Security Assessment by Fault Injection and Behavior Monitoring." In *Proc. 12th Int'l World Wide Web Conference*, p.148-159, Budapest, Hungary, 2003.

[14] Huang, Y. W., Yu, F., Hang, C., Tsai, C. H., Lee, D.T., Kuo, S. Y. "Securing Web Application Code by Static Analysis and Runtime Inspection." In: *Proc. 13th Int'l World Wide Web Conference*, New York, 2004.

[15] Hughes, F. "PHP: Most Popular Server-Side Web Scripting Technology." LWN.net. http://lwn.net/Articles/1433/

[16] Kroening, D., Strichman, O. "Efficient Computation of Recurrence Diameters." In *Proc. 4th Int'l Conf. Verification, Model Checking, and Abstract Interpretation*, p.298-309, volume LNCS 2575, New York, 2003. Springer-Verlag.

[17] Meier, J.D., Mackman, A., Vasireddy, S. Dunner, M., Escamilla, R., Murukan, A. "Improving Web Application Security—Threats and Countermeasures." Microsoft Corporation, 2003.

[18] Merzbacher, M., Patterson, D. "Measuring End-User Availability on the Web: Practical Experience." In *Proc. 2002 Int'l Conf. Dependable Systems and Networks*, p.473-488, Washington, D.C., 2002.

[19] Moskewicz, M. W., Madigan, C. F., Zhao, Y., Zhang, L., Malik, S. "Chaff: Engineering an Efficient SAT Solver." In *Proc. 38th Design Automation Conference*, session 33.1, New Orleans, LA, 2001.

[20] OWASP. "The Ten Most Critical Web Application Security Vulnerabilities." OWASP Whitepaper, version 1.0, 2003.

[21] Pottier, F., Simonet, V. "Information Flow Inference for ML." ACM Transactions on Programming Languages and Systems, 25(1):117-158, 2003.

[22] Sandhu, R. S. "Lattice-Based Access Control Models." IEEE Computer, 26(11):9-19, 1993.

[23] Sanjit, A. S., Bryant, R. E., "Unbounded, Fully Symbolic Model Checking of Timed Automata using Boolean Methods." In *Proc. 15th Int'l Conf. Computer-Aided Verification*, p.154-166, volume LNCS 2725, Boulder, Colorado, 2003. Springer-Verlag.

[24] Scott, D., Sharp, R. "Abstracting Application-Level Web Security." In *Proc. 11th Int'l World Wide Web Conference*, p.396-407, Honolulu, Hawaii, 2002.

[25] Shankar, U., Talwar, K., Foster, J. S., Wagner, D. "Detecting Format String Vulnerabilities with Type Qualifiers." In *Proc. 10th USENIX Security Symposium*, p.201-220, Washington DC, 2002.

[26] Strom, R. E., Yemini, S. A. "Typestate: A Programming Language Concept for Enhancing Software Reliability." IEEE Transactions on Software Engineering, 12(1):157-171, Jan 1986.

[27] Watts, G. "PHPXref: PHP Cross Referencing Documentation Generator." Sep 2003. http://phpxref.sourceforge.net/

[28] Woodman, S., Morgan, G., Parkin, S. "Portal Replication for Web Application Availability Via SOAP." In *Proc. 8th IEEE Int'l Workshop on Object-Oriented Real-Time Dependable Systems*, p.122-130, Guadalajara, Mexico, 2003.

[29] Wright, A. K, Cartwright, R. "A Practical Soft Type System for Scheme." ACM Transactions on Programming Languages and Systems, 19(1):87-152, Jan 1999.