HAMPI: A String Solver for Testing, Analysis and Vulnerability Detection

Vijay Ganesh[1], Adam Kieżun[2]
Shay Artzi[3], Philip J. Guo[4], Pieter Hooimeijer[5], Michael Ernst[6]

[1]Massachusetts Institute of Technology, [2]Harvard Medical School
[1] vganesh@csail.mit.edu, [2] akiezun@gmail.com
[3]IBM Research, [4]Stanford University, [5]University of Virginia, [6]University of Washington
[3]artzi@us.ibm.com, [4]pg@cs.stanford.edu, [5]pieter@cs.virginia.edu, [6]mernst@cs.washington.edu

Abstract. Many automatic testing, analysis, and verification techniques for programs can effectively be reduced to a constraint-generation phase followed by a constraint-solving phase. This separation of concerns often leads to more effective and maintainable software reliability tools. The increasing efficiency of off-the-shelf constraint solvers makes this approach even more compelling. However, there are few effective and sufficiently expressive off-the-shelf solvers for string constraints generated by analysis of string-manipulating programs, and hence researchers end up implementing their own ad-hoc solvers. Thus, there is a clear need for an effective and expressive string-constraint solver that can be easily integrated into a variety of applications.

To fulfill this need, we designed and implemented HAMPI, an efficient and easy-to-use string solver. Users of the HAMPI string solver specify constraints using membership predicate over regular expressions, context-free grammars, and equality/dis-equality between string terms. These terms are constructed out of string constants, bounded string variables, and typical string operations such as concatenation and substring extraction. HAMPI takes such a constraint as input and decides whether it is satisfiable or not. If an input constraint is satisfiable, HAMPI generates a satsfying assignment for the string variables that occur in it.

We demonstrate HAMPI's expressiveness and efficiency by applying it to program analysis and automated testing: We used HAMPI in static and dynamic analyses for finding SQL injection vulnerabilities in Web applications with hundreds of thousands of lines of code. We also used HAMPI in the context of automated bug finding in C programs using dynamic systematic testing (also known as concolic testing). HAMPI's source code, documentation, and experimental data are available at `http://people.csail.mit.edu/akiezun/hampi`.

1 Introduction

Many automatic testing [4, 9], analysis [12], and verification [14] techniques for programs can be effectively reduced to a constraint-generation phase followed by a constraint solving phase. This separation of concerns often leads to more effective and maintainable tools. Such an approach to analyzing programs is becoming more effective as off-the-shelf constraint solvers for Boolean SAT [20] and other theories [5, 8] continue to become more efficient. Most of these solvers are aimed at propositional logic, linear arithmetic, theories of functions, arrays or bit-vectors [5].

Many programs (e.g., Web applications) take string values as input, manipulate them, and then use them in sensitive operations such as database queries. Analyses of such string-manipulating programs in techniques for automatic testing [2, 6, 9], verifying correctness of program output [21], and finding security faults [25] produce *string constraints*, which are then solved by custom string solvers written by the authors of these analyses. Writing a custom solver for every application is time-consuming and error-prone, and the lack of separation of concerns may lead to systems that are difficult to maintain. Thus, there is a clear need for an effective and sufficiently expressive off-the-shelf string-constraint solver that can be easily integrated into a variety of applications.

To fulfill this need, we designed and implemented HAMPI [1], a solver for constraints over bounded string variables. HAMPI constraints express membership in bounded regular and context-free languages, substring relation, and equalities/dis-equalities over string terms.

String terms in the HAMPI language are constructed out of string constants, bounded string variables, concatenation, and sub-string extraction operations. Regular expressions and context-free grammar terms are constructed out of standard regular expression operations and grammar productions, respectively. Atomic formulas in the HAMPI language are equality over string terms, the membership predicate for regular expressions and context-free grammars, and the substring predicate that takes two string terms and asserts that one is a substring of the other. Given a set of constraints, HAMPI outputs a string that satisfies all the constraints, or reports that the constraints are unsatisfiable.

HAMPI is designed to be used as a component in testing, analysis, and verification applications. HAMPI can also be used to solve the intersection, containment, and equivalence problems for bounded regular and context-free languages.

A key feature of HAMPI is bounding of regular and context-free languages. Bounding makes HAMPI different from custom string-constraint solvers commonly used in testing and analysis tools [6]. As we demonstrate in our experiments, for many practical applications, bounding the input languages is not a handicap. In fact, it allows for a more expressive input language that enables operations on context-free languages that would be undecidable without bounding. Furthermore, bounding makes the satisfiability problem solved by HAMPI more tractable. This difference is analogous to that between model-checking and bounded model-checking [1].

As one example application, HAMPI's input language can encode constraints on SQL queries to find possible injection attacks, such as:

> Find a string v of at most 12 characters, such that the SQL query "`SELECT msg FROM messages WHERE topicid=`v" is a syntactically valid SQL statement, and that the query contains the substring "`OR 1=1`".

Note that "`OR 1=1`" is a common tautology that can lead to SQL injection attacks. HAMPI either finds a string value that satisfies these constraints or answers that no satisfying value exists. For the above example, the string "`1 OR 1=1`" is a valid solution.

HAMPI Overview: HAMPI takes four steps to solve input string constraints.

[1] This paper is an extended version of the HAMPI paper accepted at the International Symposium on Software Testing and Analysis (ISSTA) 2009 conference. A journal version is under submission.

1. Normalize the input constraints to a *core form*, which consists of expressions of the form $v \in R$ or $v \notin R$, where v is a bounded string variable, and R is a regular expression.
2. Translate core form string constraints into a quantifier-free logic of bit-vectors. A bit-vector is a bounded, ordered list of bits. The fragment of bit-vector logic that HAMPI uses allows standard Boolean operations, bit comparisons, and extracting sub-vectors.
3. Invoke the STP bit-vector solver [8] on the bit-vector constraints.
4. If STP reports that the constraints are unsatisfiable, then HAMPI reports the same. Otherwise, STP will generate a satisfying assignment in its bit-vector language, so HAMPI decodes this to output an ASCII string solution.

Experimental Results Summary: We ran four experiments to evaluate HAMPI. Our results show that HAMPI is efficient and that its input language can express string constraints that arise from real-world program analysis and automated testing tools.

1. *SQL Injection Vulnerability Detection (static analysis):* We used HAMPI in a static analysis tool [23] for identifying SQL injection vulnerabilities. We applied the analysis tool to 6 PHP Web applications (total lines of code: 339,750). HAMPI solved all constraints generated by the analysis, and solved 99.7% of those constraints in less than 1 second per constraint. All solutions found by HAMPI for these constraints were less than 5 characters long. These experiments bolster our claim that bounding the string constraints is not a handicap.
2. *SQL Injection Attack Generation (dynamic analysis):* We used HAMPI in Ardilla, a dynamic analysis tool for creating SQL injection attacks [17]. We applied Ardilla to 5 PHP Web applications (total lines of code: 14,941). HAMPI successfully replaced a custom-made attack generator and constructed all 23 attacks on those applications that Ardilla originally constructed.
3. *Input Generation for Systematic Testing:* We used HAMPI in Klee [3], a systematic-testing tool for C programs. We applied Klee to 3 programs with structured input formats (total executable lines of code: 4,100). We used HAMPI to generate constraints that specify legal inputs to these programs. HAMPI's constraints eliminated all illegal inputs, improved the line-coverage by up to 2× overall (and up to 5× in parsing code), and discovered 3 new error-revealing inputs.

1.1 PAPER ORGANIZATION

We first introduce HAMPI's capabilities with an example (§2), then present HAMPI's input format and solving algorithm (§3), and present experimental evaluation (§4). We briefly touch upon related work in (§5).

2 Example: SQL Injection

SQL injections are a prevalent class of Web-application vulnerabilities. This section illustrates how an automated tool [17, 25] could use HAMPI to detect SQL injection vulnerabilities and to produce attack inputs.

```
1 $my_topicid = $_GET['topicid'];
2
3 $sqlstmt = "SELECT msg FROM messages WHERE topicid='$my_topicid'";
4 $result = mysql_query($sqlstmt);
5
6 //display messages
7 while($row = mysql_fetch_assoc($result)){
8   echo "Message " . $row['msg'];
9 }
```

Fig. 1. Fragment of a PHP program that displays messages stored in a MySQL database. This program is vulnerable to an SQL injection attack. Section 2 discusses the vulnerability.

```
1  //string variable representing '$my_topicid' from Figure 1
2  var v:6..12;   // size is between 6 and 12 characters
3
4  //simple SQL context-free grammar
5  cfg SqlSmall := "SELECT " (Letter)+ " FROM " (Letter)+ " WHERE " Cond;
6  cfg Cond     := Val "=" Val | Cond " OR " Cond";
7  cfg Val      := (Letter)+ | "'" (LetterOrDigit)* "'" | (Digit)+;
8  cfg LetterOrDigit := Letter | Digit;
9  cfg Letter := ['a'-'z'] ;
10 cfg Digit  := ['0'-'9'] ;
11
12 //the SQL query $sqlstmt from line 3 of Figure 1
13 val q := concat("SELECT msg FROM messages WHERE topicid='", v, "'");
14
15 //constraint conjuncts
16 assert q in SqlSmall;
17 assert q contains "OR '1'='1'";
```

Fig. 2. HAMPI input that, when solved, produces an SQL injection attack vector for the vulnerability from Figure 1.

Figure 1 shows a fragment of a PHP program that implements a simple Web application: a message board that allows users to read and post messages stored in a MySQL database. Users of the message board fill in an HTML form (not shown here) that communicates the inputs to the server via a specially formatted URL, e.g., `http://www.mysite.com/?topicid=1`. Input parameters passed inside the URL are available in the `$_GET` associative array. In the above example URL, the input has one key-value pair: `topicid=1`. The program fragment in Figure 1 retrieves and displays messages for the given topic.

This program is vulnerable to an SQL injection attack. An attacker can read all messages in the database (including ones intended to be private) by crafting a malicious URL like:

```
http://www.mysite.com/?topicid=1' OR '1'='1
```

Upon being invoked with that URL, the program reads the string

```
1' OR '1'='1
```

as the value of the `$my_topicid` variable, constructs an SQL query by concatenating it to a constant string, and submits the following query to the database in line 4:

```
SELECT msg FROM messages WHERE topicid='1' OR '1'='1'
```

The `WHERE` condition is always true because it contains the tautology `'1'='1'`. Thus, the query retrieves all messages, possibly leaking private information.

A programmer or an automated tool might ask, "Can an attacker exploit the `topicid` parameter and introduce a `OR '1'='1'` tautology into a syntactically-correct SQL query at line 4 in the code of Figure 1?" The HAMPI solver answers such questions and creates strings that can be used as exploits.

The HAMPI constraints in Figure 2 formalize the question in our example. Automated vulnerability-scanning tools [17, 25] can create HAMPI constraints via either static or dynamic program analysis (we demonstrate both static and dynamic techniques in our evaluation in Sections 4.1 and 4.2, respectively). Specifically, a tool could create the HAMPI input shown in Figure 2 by analyzing the code of Figure 1.

We now discuss various features of the HAMPI input language that Figure 2 illustrates. (Section 3.1 describes the input language in more detail.)

- Keyword `var` (line 2) introduces a *string variable* `v`. The variable has a size in the range of 6 to 12 characters. The goal of the HAMPI solver is to find a string that, when assigned to the string variable, satisfies all the constraints. In this example, HAMPI will search for solutions of sizes between 6 and 12.
- Keyword `cfg` (lines 5–10) introduces a *context-free grammar*, for a fragment of the SQL grammar of `SELECT` statements.
- Keyword `val` (line 13) introduces a temporary variable `q`, declared as a *concatenation* of constant strings and the string variable `v`. This variable represents an SQL query corresponding to the PHP `$sqlstmt` variable from line 3 in Figure 1.
- Keyword `assert` defines a constraint. The top-level HAMPI constraint is a conjunction of `assert` statements. Line 16 specifies that the query string `q` must be a member of the context-free language `SqlSmall` (syntactically-correct SQL). Line 17 specifies that the variable `v` must contain a specific substring (e.g., the `OR '1'='1'` tautology that can lead to an SQL injection attack).

HAMPI can solve the constraints specified in Figure 2 and find a value for `v` such as

```
1' OR '1'='1
```

which is a value for `$_GET['topicid']` that can lead to an SQL injection attack.

3 The Hampi String Constraint Solver

HAMPI finds a string that satisfies constraints specified in the input, or decides that no satisfying string exists. HAMPI works in four steps, as illustrated in Figure 3:

1. Normalize the input constraints to a *core form* (§3.2).
2. Encode core form constraints in bit-vector logic (§3.3).
3. Invoke the STP solver [8] on the bit-vector constraints (§3.3).
4. Decode the results obtained from STP (§3.3).

Users can invoke HAMPI with a text-based command-line front-end (using the input grammar in Figure 4) or with a Java API to directly construct the HAMPI constraints.

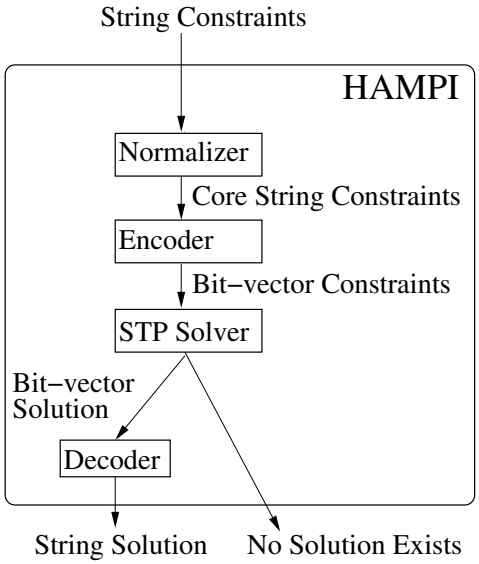

Fig. 3. Schematic view of the HAMPI string constraint solver. Input enters at the top, and output exits at the bottom. Section 3 describes the HAMPI solver.

3.1 HAMPI Input Language for String Constraints

We now discuss the salient features of HAMPI's input language (Figure 4) and illustrate them on examples. The language is expressive enough to encode string constraints generated by typical program analysis, testing, and security applications. HAMPI's language supports declaration of bounded string variables and constants, concatenation and extraction operation over string terms, equality over string terms, regular-language operations, membership predicate, and declaration of context-free and regular languages, temporaries and constraints.

Declaration of String Variable (`var` keyword) A HAMPI input must declare a *single* string variable and specify its size range as lower and upper bounds on the number of characters. If the input constraints are satisfiable, then HAMPI finds a value for the variable that satisfies all constraints. For example, the following line declares a string variable named `v` with a size between 5 and 20 characters:

```
var v:5..20;
```

Extraction Operation HAMPI supports extraction of substrings from string terms (as shown in Figure 4). An example of extraction operation is as follows:

```
var longv:20;
val v1 := longv[0:9];
```

where 0 is the offset (or starting character of the extraction operation), and 9 is the length of the resultant string, in terms of the number of characters of *longv*.

Input	::= *Var Stmt*∗	HAMPI input (with a single string variable)
Var	::= **var** *Id* : *Int*..*Int*	string variable (length lower..upper bound)
Stmt	::= *Cfg* | *Reg* | *Val* | *Assert*	statement
Cfg	::= **cfg** *Id* := *CfgProdRHS*	context-free language
CfgProdRHS	::= *CFG declaration in EBNF*	Extended Backus-Naur Form (EBNF)
Reg	::= **reg** *Id* := *RegElem*	regular-language
RegElem	::= *StrConst*	string constant
	| *Id*	variable reference
	| **fixsize**(*Id* , *Int*)	CFG fixed-sizing
	| **or**(*RegElem* ∗)	union
	| **concat**(*RegElem* ∗)	concatenation
	| **star**(*RegElem*)	Kleene star
Val	::= **val** *Id* :=*ValElem*	temporary variable
ValElem	::= *Id*	
	| *StrConst*	
	| **concat**(*ValElem* ∗)	concatenation
	| *ValElem*[*offset* : *length*]	extraction(ValElem, offset, length)
Assert	::= **assert** *Id* [**not**]? **in** *Reg*	regular-language membership
	| **assert** *Id* [**not**]? **in** *Cfg*	context-free language membership
	| **assert** *Id* [**not**]? **contains** *StrConst*	substring
	| **assert** *Id* [**not**]? = *Id*	word equation (equality/dis-equality)
Id	::= *String identifier*	
StrConst	::= "*String literal constant*"	
Int	::= *Non-negative integer*	

Fig. 4. Summary of HAMPI's input language. **Terminals** are bold-faced, *nonterminals* are italicized. A HAMPI input (*Input*) is a variable declaration, followed by a list of these statements: context-free-grammar declarations, regular-language declarations, temporary variables, and assertions.

Declaration of Multiple Variables The user can simulate having multiple variables by declaring a single long string variable and using the extract operation: Disjoint extractions of the single long variable can act as multiple variables. For example, to declare two string variables of length 10 named `v1` and `v2`, use:

```
var longv:20;
val v1 := longv[0:9];
val v2 := longv[10:9];
```

The `val` keyword declares a temporary (derived) variable and will be described later in this section.

Declarations of Context-free Languages (`cfg` keyword) HAMPI input can declare context-free languages using grammars in the standard notation: Extended Backus-Naur Form (EBNF). Terminals are enclosed in double quotes (e.g., `"SELECT"`), and productions are separated by the vertical bar symbol (`|`). Grammars may contain special symbols for repetition (`+` and `*`) and character ranges (e.g., `[a-z]`). For example,

lines 5–10 in Figure 2 show the declaration of a context-free grammar for a subset of SQL.

HAMPI's format for context-free grammars is as expressive as that of widely-used tools such as Yacc/Lex; in fact, we have written a simple syntax-driven script that transforms a Yacc specification to HAMPI format (available on the HAMPI website). HAMPI can only solve constraints over bounded context-free grammars. However, the user does not have to manually specify bounds, since HAMPI automatically derives a bound by analyzing the bound on the input string variable and the longest possible string that can be constructed out of concatenation and extraction operations.

Declarations of Regular Languages (`reg` keyword) HAMPI input can declare regular languages using the following regular expressions: (i) a singleton set with a string constant, (ii) a concatenation/union of regular languages, (iii) a repetition (Kleene star) of a regular language, (iv) bounding of a context-free language, which HAMPI does automatically. Every regular language can be expressed using the first three of those operations [22].

For example, `(b*ab*ab*)*` is a regular expression that describes the language of strings over the alphabet {`a`,`b`}, with an even number of `a` symbols. In HAMPI syntax this is:

```
reg Bstar := star("b");            // 'helper' expression
reg EvenA := star(concat(Bstar, "a", Bstar, "a", Bstar));
```

The HAMPI website contains a script to convert Perl Compatible Regular Expressions (PCRE) into HAMPI syntax. Also note that context-free grammars in HAMPI are implicitly bounded, and hence are regular expressions.

Temporary Declarations (`val` keyword) Temporary variables are shortcuts for expressing constraints on expressions that are concatenations of the string variable and constants or extractions. For example, line 13 in Figure 2 declares a temporary variable named `q` by concatenating two constant strings to the variable `v`:

```
val q := concat("SELECT msg FROM messages WHERE topicid='", v, "'");
```

Constraints (`assert` keyword) HAMPI constraints specify membership of variables in regular and context-free languages, substrings, and word equations. HAMPI solves for the conjunction of all constraints listed in the input.

- Membership Predicate (`in`): Assert that a variable is in a context-free or regular language. For example, line 16 in Figure 2 declares that the string value of the temporary variable `q` is in the context-free language `SqlSmall`:

  ```
  assert q in SqlSmall;
  ```

- Substring Relation (`contains`): Assert that a variable contains the given string constant. For example, line 17 in Figure 2 declares that the string value of the temporary variable `q` contains an SQL tautology:

  ```
  assert q contains "OR '1'='1'";
  ```

S	::=	*Constraint*	
	|	$S \wedge Constraint$	conjunction
Constraint	::=	$StrExp \in RegExp$	membership
	|	$StrExp \notin RegExp$	non-membership
Constraint	::=	$StrExp = StrExp$	equality
	|	$StrExp \neq StrExp$	dis-equality
StrExp	::=	*Var*	input variable
	|	*StrConst*	string constant
	|	*StrExp StrExp*	concatenation
	|	*StrExp*[*offset* : *length*]	extraction
RegExp	::=	*StrConst*	constant
	|	*RegExp* + *RegExp*	union
	|	*RegExp RegExp*	concatenation
	|	*RegExp*$\star$	star

Fig. 5. The grammar of core form string constraints. *Var*, *StrConst*, and *Int* are defined in Figure 4.

– String Equalities (=): Asserts that two string terms are equal (also known as word equations). In HAMPI, both sides of the equality must ultimately originate from the same single string variable. For example, the `extract` operator can assert that two portions of a string must be equal:

```
var v:20;
val v1 := v[0:9];
val v2 := v[10:9];
assert v1 = 2v;
```

All of these constraints may be negated by preceding them with a `not` keyword.

3.2 Core Form of String Constraints

After parsing and checking the input, HAMPI normalizes the string constraints to a core form. The core form (grammar shown in Figure 5) is an internal intermediate representation that is easier than raw HAMPI input to encode in bit-vector logic. A core form string constraint specifies membership (or its negation) in a regular language: $StrExp \in RegExp$ or $StrExp \notin RegExp$, where *StrExp* is an expression composed of concatenations of string constants, extractions, and occurrences of the (sole) string variable, and *RegExp* is a regular expression.

HAMPI normalizes its input into core form in 3 steps:

1. Expand all temporary variables, i.e., replace each reference to a temporary variable with the variable's definition (HAMPI forbids recursive definitions of temporaries).
2. Calculate maximum size and bound all context-free grammar expressions into regular expressions (see below for the algorithm).
3. Expand all regular-language declarations, i.e., replace each reference to a regular-language variable with the variable's definition.

Bounding of Context-free Grammars: HAMPI uses the following algorithm to create regular expressions that specify the set of strings of a fixed length that are derivable from a context-free grammar:

1. Expand all special symbols in the grammar (e.g., repetition, option, character range).
2. Remove ϵ productions [22].
3. Construct the regular expression that encodes all bounded strings of the grammar as follows: First, pre-compute the length of the shortest and longest (if exists) string that can be generated from each nonterminal (i.e., lower and upper bounds). Second, given a size n and a nonterminal N, examine all productions for N. For each production $N ::= S_1 \ldots S_k$, where each S_i may be a terminal or a nonterminal, enumerate all possible partitions of n characters to k grammar symbols (HAMPI takes the pre-computed lower and upper bounds to make the enumeration more efficient). Then, create the sub-expressions recursively and combine the subexpressions with a concatenation operator. Memoization of intermediate results makes this (worst-case exponential in k) process scalable.

Here is an example of grammar fixed-sizing: Consider the following grammar of well-balanced parentheses and the problem of finding the regular language that consists of all strings of length 6 that can be generated from the nonterminal `E`.

```
cfg E := "()" | E E | "(" E ")" ;
```

The grammar does not contain special symbols or ϵ productions, so first two steps of the algorithm do nothing. Then, HAMPI determines that the shortest string `E` can generate is of length 2. There are three productions for the nonterminal `E`, so the final regular expression is a union of three parts. The first production, `E := "()"`, generates no strings of size 6 (and only one string of size 2). The second production, `E := E E`, generates strings of size 6 in two ways: either the first occurrence of `E` generates 2 characters and the second occurrence generates 4 characters, or the first occurrence generates 4 characters and the second occurrence generates 2 characters. From the pre-processing step, HAMPI knows that the only other possible partition of 6 characters is 3–3, which HAMPI tries and fails (because `E` cannot generate 3-character strings). The third production, `E := "(" E ")"`, generates strings of size 6 in only one way: the nonterminal `E` must generate 4 characters. In each case, HAMPI creates the sub-expressions recursively. The resulting regular expression for this example is (plus signs denote union and square brackets group sub-expressions):

$$()[()() + (())] \;+\; [()() + (())]() \;+\; ([()() + (())])$$

3.3 Bit-vector Encoding and Solving

HAMPI encodes the core form string constraints as formulas in the logic of fixed-size bit-vectors. A bit-vector is a fixed-size, ordered list of bits. The fragment of bit-vector logic that HAMPI uses contains standard Boolean operations, extracting sub-vectors, and comparing bit-vectors (We refer the reader to [8] for a detailed description of the bit-vector logic used by HAMPI) HAMPI asks the STP bit-vector solver [8] for a satisfying assignment to the resulting bit-vector formula. If STP finds an assignment, HAMPI decodes it, and produces a string solution for the input constraints. If STP cannot find a solution, HAMPI terminates and declares the input constraints unsatisfiable.

Every core form string constraint is encoded separately, as a conjunct in a bit-vector logic formula. HAMPI encodes the core form string constraint $StrExp \in RegExp$ recursively, by case analysis of the regular expression $RegExp$, as follows:

– HAMPI encodes constants by enforcing constant values in the relevant elements of the bit-vector variable (HAMPI encodes characters using 8-bit ASCII codes).
– HAMPI encodes the union operator (+) as a disjunction in the bit-vector logic.
– HAMPI encodes the concatenation operator by enumerating all possible distributions of the characters to the sub-expressions, encoding the sub-expressions recursively, and combining the sub-formulas in a conjunction.
– HAMPI encodes the ⋆ similarly to concatenation — a star is a concatenation with variable number of occurrences. To encode the star, HAMPI finds the upper bound on the number of occurrences (the number of characters in the string is always a sound upper bound).

After STP finds a solution to the bit-vector formula (if one exists), HAMPI decodes the solution by reading 8-bit sub-vectors as consecutive ASCII characters.

3.4 Example of HAMPI Constraint Solving

We now illustrate the entire constraint solving process end-to-end on a simple example. Given the following input:

```
var v:2..2; // fixed-size string of length 2
cfg E := "()" | E E | "(" E ")";
reg Efixed := fixsize(E, 6);
val q := concat( "((" , v , "))" );
assert q in Efixed;    // turns into constraint c1
assert q contains "())"; // turns into constraint c2
```

HAMPI tries to find a satisfying assignment for variable `v` by following the four-step algorithm[2] in Figure 3:

Step 1. Normalize constraints to core form, using the algorithm in Section 3.2:

c1 [`assert q in Efixed`]: $((v)) \in ()[()() + (())] + [()() + (())]() + ([()() + (())])$

c2 [`assert q contains "())"`]: $((v)) \in [(+)]\star\ ())\ [(+)]\star$

Step 2. Encode the core-form constraints in bit-vector logic. We show how HAMPI encodes constraint **c1**; the process for **c2** is similar. HAMPI creates a bit-vector variable *bv* of length 6*8=48 bits, to represent the left-hand side of **c1** (since `Efixed` is 6 bytes). Characters are encoded using ASCII codes: '`(`' is 40 in ASCII, and '`)`' is 41. HAMPI encodes the left-hand-side expression of **c1**, `(( v ))`, as formula L_1, by specifying the constant values:

$$L_1 : (bv[0] = 40) \wedge (bv[1] = 40) \wedge (bv[4] = 41) \wedge (bv[5] = 41)$$

[2] The alphabet of the regular expression or context-free grammar in a HAMPI input is implicitly restricted to the terminals specified

Bytes $bv[2]$ and $bv[3]$ are reserved for v, a 2-byte variable. The top-level regular expression in the right-hand side of **c1** is a 3-way union, so the result of the encoding is a 3-way disjunction. For the first disjunct `()[()() + (())]`, HAMPI creates the following formula D_{1a}:

$$bv[0] = 40 \wedge bv[1] = 41 \wedge$$
$$\Big(\big(bv[2] = 40 \wedge bv[3] = 41 \wedge bv[4] = 40 \wedge bv[5] = 41\big) \vee$$
$$\big(bv[2] = 40 \wedge bv[3] = 40 \wedge bv[4] = 41 \wedge bv[5] = 41\big)\Big)$$

Formulas D_{1b} and D_{1c} for the remaining conjuncts are similar. The bit-vector formula that encodes **c1** is

$$C_1 = L_1 \wedge \big(D_{1a} \vee D_{1b} \vee D_{1c}\big)$$

Similarly, a formula C_2 (not shown here) encodes **c2**. The formula that HAMPI sends to the STP solver is

$$(C_1 \wedge C_2)$$

Step 3. STP finds a solution that satisfies the formula:

$$bv[0] = 40, bv[1] = 40, bv[2] = 41, bv[3] = 40, bv[4] = 41, bv[5] = 41$$

In decoded ASCII, the solution is "`(()())`" (quote marks not part of solution string).

Step 4. HAMPI reads the assignment for variable v off of the STP solution, by decoding the elements of bv that correspond to v, i.e., elements 2 and 3. HAMPI reports the solution for v as "`)(`". String "`()`" is another legal solution for v, but STP only finds one solution.

4 Evaluation

We experimentally tested HAMPI's applicability to practical problems involving string constraints and compared HAMPI's performance and scalability to another string-constraint solver. We ran the following four experiments:

1. We used HAMPI in a static-analysis tool [23] that identifies possible SQL injection vulnerabilities (Section 4.1).
2. We used HAMPI in Ardilla [17], a dynamic-analysis tool that creates SQL injection attacks (Section 4.2).
3. We used HAMPI in Klee, a systematic testing tool for C programs (Section 4.3).

Unless otherwise noted, we ran all experiments on a 2.2GHz Pentium 4 PC with 1 GB of RAM running Debian Linux, executing HAMPI on Sun Java Client VM 1.6.0-b105 with 700MB of heap space. We ran HAMPI with all optimizations on, but flushed

the whole internal state after solving each input to ensure fairness in timing measurements, i.e., preventing artificially low runtimes when solving a series of structurally-similar inputs. The results of our experiments demonstrate that HAMPI is expressive in encoding real constraint problems that arise in security analysis and automated testing, that it can be integrated into existing testing tools, and that it can efficiently solve large constraints obtained from real programs. HAMPI's source code and documentation, experimental data, and additional results are available at `http://people.csail.mit.edu/akiezun/hampi`.

4.1 Identifying SQL Injection Vulnerabilities Using Static Analysis

We evaluated HAMPI's applicability to finding SQL injection vulnerabilities in the context of a static analysis. We used the tool from Wassermann and Su [23] that, given source code of a PHP Web application, identifies potential SQL injection vulnerabilities. The tool computes a context-free grammar G that conservatively approximates all string values that can flow into each program variable. Then, for each variable that represents a database query, the tool checks whether $L(G) \cap L(R)$ is empty, where $L(R)$ is a regular language that describes undesirable strings or attack vectors (strings that can exploit a security vulnerability). If the intersection is empty, then Wassermann and Su's tool reports the program to be safe. Otherwise, the program may be vulnerable to SQL injection attacks.

An example $L(R)$ that Wassermann and Su use — the language of strings that contain an odd number of unescaped single quotes — is given by the regular expression (we used this R in our experiments):

```
R = (([^']|\')*[^\])?'
    (([^']|\')*[^\])?'
     (([^']|\')*[^\])?'([^']|\')*
```

Using HAMPI in such an analysis offers two important advantages. First, it eliminates a time-consuming and error-prone reimplementation of a critical component: the string-constraint solver. To compute the language intersection, Wassermann and Su implemented a custom solver based on the algorithm by Minamide [19]. Second, HAMPI creates concrete example strings from the language intersection, which is important for generating attack vectors; Wassermann and Su's custom solver only checks for emptiness of the intersection, and does not create example strings.

Using a fixed-size string-constraint solver, such as HAMPI, has its limitations. An advantage of using an unbounded-length string-constraint solver is that if the solver determines that the input constraints have no solution, then there is indeed no solution. In the case of HAMPI, however, we can only conclude that there is no solution of the given size.

Experiment: We performed the experiment on 6 PHP applications. Of these, 5 were applications used by Wassermann and Su to evaluate their tool [23]. We added 1 large application (`claroline`, a builder for online education courses, with 169 kLOC) from another paper by the same authors [24]. Each of the applications has known SQL injection vulnerabilities. The total size of the applications was 339,750 lines of code.

Wassermann and Su's tool found 1,367 opportunities to compute language intersection, each time with a different grammar G (built from the static analysis) but with the same regular expression R describing undesirable strings. For each input (i.e., pair of G and R), we used both Hampi and Wassermann and Su's custom solver to compute whether the intersection $L(G) \cap L(R)$ was empty.

When the intersection is *not* empty, Wassermann and Su's tool cannot produce an example string for those inputs, but Hampi can. To do so, we varied the size N of the string variable between 1 and 15, and for each N, we measured the total Hampi solving time, and whether the result was UNSAT or a satisfying assignment.

Results: We found empirically that when a solution exists, it can be very short. In 306 of the 1,367 inputs, the intersection was *not* empty (both solvers produced identical results). Out of the 306 inputs with non-empty intersections, we measured the percentage for which Hampi found a solution (for increasing values of N): 2% for $N = 1$, 70% for $N = 2$, 88% for $N = 3$, and 100% for $N = 4$. That is, in this large dataset, all non-empty intersections contain strings with no longer than 4 characters. Due to false positives inherent in Wassermann and Su's static analysis, the strings generated from the intersection do not necessarily constitute real attack vectors. However, this is a limitation of the static analysis, not of Hampi.

We measured how Hampi's solving time depends on the size of the grammar. We measured the size of the grammar as the sum of lengths of all productions (we counted ϵ-productions as of length 1). Among the 1,367 grammars in the dataset, the mean size was 5490.5, standard deviation 4313.3, minimum 44, maximum 37955. We ran Hampi for $N = 4$, i.e., the length at which all satisfying assignments were found. Hampi solves most of these queries quickly (99.7% in less than 1 second, and only 1 query took 10 seconds).

4.2 Creating SQL Injection Attacks Using Dynamic Analysis

We evaluated Hampi's ability to automatically find SQL injection attack strings using constraints produced by running a dynamic-analysis tool on PHP Web applications. For this experiment, we used Ardilla [17], a tool that constructs SQL injection and Cross-site Scripting (XSS) attacks by combining automated input generation, dynamic tainting, and generation and evaluation of candidate attack strings.

One component of Ardilla, the *attack generator*, creates candidate attack strings from a pre-defined list of attack patterns. Though its pattern list is extensible, Ardilla's attack generator is neither targeted nor exhaustive: The generator does not attempt to create valid SQL statements but rather simply assigns pre-defined values from the attack patterns list one-by-one to variables identified as vulnerable by the dynamic tainting component; it does so until an attack is found or until there are no more patterns to try.

For this experiment, we replaced the attack generator with the Hampi string solver. This reduces the problem of finding SQL injection attacks to one of string constraint generation followed by string constraint solving. This replacement makes attack creation targeted and exhaustive — Hampi constraints encode the SQL grammar and, if there is an attack of a given length, Hampi is sure to find it.

To use Hampi with Ardilla, we also replaced Ardilla's dynamic tainting component with a concolic execution [10] component. This required code changes were quite ex-

tensive but fairly standard. Concolic execution creates and maintains symbolic expressions for each concrete runtime value derived from the input. For example, if a value is derived as a concatenation of user-provided parameter `p` and a constant string `"abc"`, then its symbolic expression is `concat(p, "abc")`. This component is required to generate the constraints for input to HAMPI.

The HAMPI input includes a partial SQL grammar (similar to that in Figure 2). We wrote a grammar that covers a subset of SQL queries commonly observed in Web applications, which includes `SELECT`, `INSERT`, `UPDATE`, and `DELETE`, all with `WHERE` clauses. The grammar has size is 74, according to the metric of Section 4.1. Each terminal is represented by a single unique character.

We ran our modified Ardilla on 5 PHP applications (the same set as the original Ardilla study [17], totaling 14,941 lines of PHP code). The original study identified 23 SQL injection vulnerabilities in these applications. Ardilla generated 216 HAMPI inputs, each of which is a string constraint built from the execution of a particular path through an application. For each constraint, we used HAMPI to find an attack string of size $N \leq 6$ — a solution corresponds to the value of a vulnerable PHP input parameter. Following previous work [7, 13], the generated constraint defined an attack as a syntactically valid (according to the grammar) SQL statement with a tautology in the `WHERE` clause, e.g., `OR 1=1`. We used 4 tautology patterns, distilled from several security lists[3]. We separately measured solving time for each tautology and each choice of N. A security-testing tool like Ardilla might search for the shortest attack string for *any* of the specified tautologies.

4.3 Systematic Testing of C Programs

We combined HAMPI with a state-of-the-art systematic testing tool, Klee [3], to improve Klee's ability to create valid test cases for programs that accept highly structured string inputs. Automatic test-case generation tools that use combined concrete and symbolic execution, also known as *concolic execution* [4, 11, 15] have trouble creating test cases that achieve high coverage for programs that expect structured inputs, such as those that require input strings from a context-free grammar [9, 18]. The parser components of programs that accept structured inputs (especially those auto-generated by tools such as Yacc) often contain complex control-flow with many error paths; the vast majority of paths that automatic testers explore terminate in parse errors, thus creating inputs that do not lead the program past the initial parsing stage.

Testing tools based on concolic execution mark the target program's input string as totally unconstrained (i.e., *symbolic*) and then build up constraints on the input based on the conditions of branches taken during execution. If there were a way to constrain the symbolic input string so that it conforms to a target program's specification (e.g., a context-free grammar), then the testing tool would only explore non-error paths in the program's parsing stage, thus resulting in generated inputs that reach the program's core functionality.

To demonstrate the feasibility of this technique, we used HAMPI to create grammar-based input constraints and then fed those into Klee [3] to generate test cases for C

[3] `http://www.justinshattuck.com/2007/01/18/mysql-injection-cheat-sheets`, `http://ferruh.mavituna.com/sql-injection-cheatsheet-oku`, `http://pentestmonkey.net/blog/mysql-sql-injection-cheat-sheet`

`cueconvert` (939 ELOC, 28-byte input)	symbolic	symbolic + grammar	combined
% total line coverage:	32.2%	51.4%	56.2%
% parser file line coverage (48 lines):	20.8%	77.1%	79.2%
# legal inputs / # generated inputs (%):	0 / 14 (0%)	146 / 146 (100%)	146 / 160 (91%)

`logictree` (1,492 ELOC, 7-byte input)	symbolic	symbolic + grammar	combined
% total line coverage:	31.2%	63.3%	66.8%
% parser file line coverage (17 lines):	11.8%	64.7%	64.7%
# legal inputs / # generated inputs (%):	70 / 110 (64%)	98 / 98 (100%)	188 / 208 (81%)

`bc` (1,669 ELOC, 6-byte input)	symbolic	symbolic + grammar	combined
% total line coverage:	27.1%	43.0%	47.0%
% parser file line coverage (332 lines):	11.8%	39.5%	43.1%
# legal inputs / # generated inputs (%):	2 / 27 (5%)	198 / 198 (100%)	200 / 225 (89%)

Table 1. The result of using HAMPI grammars to improve coverage of test cases generated by the Klee systematic testing tool. ELOC lists *Executable* Lines of Code, as counted by `gcov` over all `.c` files in program (whole-project line counts are several times larger, but much of that code does not directly execute). Each trial was run for 1 hour. To create minimal test suites, Klee only generates a new input when it covers new lines that previous inputs have not yet covered; the total number of explored paths is usually 2 orders of magnitude greater than the number of generated inputs. Column symbolic shows results for runs of Klee without a HAMPI grammar. Column symbolic + grammar shows results for runs of Klee with a HAMPI grammar. Column combined shows accumulated results for both kinds of runs. Section 4.3 describes the experiment.

---

programs. We compared the coverage achieved and numbers of legal (and rejected) inputs generated by running Klee with and without the HAMPI constraints.

Similar experiments have been performed by others [9, 18], and we do not claim novelty for the experimental design. However, previous studies used custom-made string solvers, while we applied HAMPI as an "off-the-shelf" solver without modifying Klee. Klee provides an API for target programs to mark inputs as symbolic and to place constraints on them. The code snippet below uses `klee_assert` to impose the constraint that all elements of `buf` must be numeric before the target program runs:

```
char buf[10]; // program input
klee_make_symbolic(buf, 10); // make all 10 bytes symbolic

// constrain buf to contain only decimal digits
for (int i = 0; i < 10; i++)
  klee_assert(('0' <= buf[i]) && (buf[i] <= '9'));

run_target_program(buf); // run target program with buf as input
```

HAMPI simplifies writing input-format constraints. Simple constraints, such as those above, can be written by hand, but it is infeasible to manually write more complex constraints for specifying, for example, that `buf` must belong to a particular context-free language. We use HAMPI to automatically compile such constraints from a grammar down to C code, which can then be fed into Klee.

We chose 3 open-source programs that specify expected inputs using context-free grammars in Yacc format (a subset of those used by Majumdar and Xu [18]). `cueconvert`

converts music playlists from `.cue` format to `.toc` format. `logictree` is a solver for propositional logic formulas. `bc` is a command-line calculator and simple programming language. All programs take input from `stdin`; Klee allows the user to create a fixed-size symbolic buffer to simulate `stdin`, so we did not need to modify these programs. For each target program, we ran the following experiment on a 3.2 GHz Pentium 4 PC with 1 GB of RAM running Fedora Linux:

1. Automatically convert its Yacc specification into HAMPI's input format (described in Section 3.1), using a script we wrote. To simplify lexical analysis, we used either a single letter or numeric digit to represent certain tokens, depending on its Lex specification (this should not reduce coverage in the parser).
2. Add a fixed-size restriction to limit the input to N bytes. Klee (similarly to, for example, SAGE [11]) actually requires a fixed-size input, which matches well with HAMPI's fixed-size input language. We empirically picked N as the largest input size for which Klee does not run out of memory. We augmented the HAMPI input to allow for strings with arbitrary numbers of trailing spaces, so that we can generate program inputs *up to* size N.
3. Run HAMPI to compile the input grammar file into STP bit-vector constraints (described in Section 3.3).
4. Automatically convert the STP constraints into C code that expresses the equivalent constraints using C variables and calls to `klee_assert()`, with a script we wrote (the script performs only simple syntactic transformations since STP operators map directly to C operators).
5. Run Klee on the target program using an N-byte input buffer, first marking that buffer as symbolic, then executing the C code that imposes the input constraints, and finally executing the program itself.
6. After a 1-hour time-limit expires, collect all generated inputs and run them through the original program (compiled using `gcov`) to measure coverage and legality of each input.
7. As a control, run Klee for 1 hour using an N-byte symbolic input buffer (with no initial constraints), collect test cases, and run them through the original program to measure coverage and legality of each input.

Table 1 summarizes our experimental setup and results. We made 3 sets of measurements: total line coverage, line coverage in the Yacc parser file that specifies the grammar rules alongside C code snippets denoting parsing actions, and numbers of inputs (test cases) generated, as well as how many of those inputs were *legal* (i.e., not rejected by the program as a parse error).

The run times for converting each Yacc grammar into HAMPI format, fixed-sizing to N bytes, running HAMPI on the fixed-size grammar, and converting the resulting STP constraints into C code are negligible; together, they took less than 1 second for each of the 3 programs. Using HAMPI in Klee improved coverage. Constraining the inputs using a HAMPI grammar resulted in up to 2× improvement in total line coverage and up to 5× improvement in line coverage within the Yacc parser file. Also, as expected, it eliminated all illegal inputs.

Using *both* sets of inputs (combined column) improved upon the coverage achieved using the grammar by up to 9%. Upon manual inspection of the extra lines covered,

we found that it was due to the fact that the runs with and without the grammar covered non-overlapping sets of lines: The inputs generated by runs without the grammar (symbolic column) covered lines dealing with processing parse errors, whereas the inputs generated with the grammar (symbolic + grammar column) never had parse errors and covered core program logic. Thus, combining test suites is useful for testing both error and regular execution paths.

With HAMPI's help, Klee uncovered more errors. Using the grammar, Klee generated 3 distinct inputs for `logictree` that uncovered (previously unknown) errors where the program entered an infinite loop. We do not know how many distinct errors these inputs identify. Without the grammar, Klee was not able to generate those same inputs within the 1-hour time limit; given the structured nature of those inputs (e.g., one is "`@x $y z`"), it is unlikely that Klee would be able to generate them within any reasonable time bound without a grammar.

We manually inspected lines of code that were not covered by any strategy. We discovered two main hindrances to achieving higher coverage: First, the input sizes were still too small to generate longer productions that exercised more code, especially problematic for the playlist files for `cueconvert`; this is a limitation of Klee running out of memory and not of HAMPI. Second, while grammars eliminated all parse errors, many generated inputs still contained *semantic* errors, such as malformed `bc` expressions and function definitions (again, unrelated to HAMPI).

5 Related Work

Decision procedures have received widespread attention within the context of program analysis, testing, and verification. Decision procedures exist for theories such as Boolean satisfiability [20] and bit-vectors [8]. In contrast, until recently there has been relatively little work on practical and expressive solvers that reason about strings or sets of strings directly. Since this is a tutorial paper we do not discuss related work in detail. Instead we point the reader to our ISSTA 2009 paper [16] for a detailed overview of previous work on decision procedures for theories of strings and practical string solvers.

References

1. A. Biere, A. Cimatti, E. Clarke, O. Strichman, and Y. Zhu. Bounded model checking. *Advances in Computers*, 58:117–148, 2003.
2. N. Bjørner, N. Tillmann, and A. Voronkov. Path feasibility analysis for string-manipulating programs. In *International Conference on Tools and Algorithms for the construction and Analysis of Systems*, York, UK, 2009. Springer Verlag.
3. C. Cadar, D. Dunbar, and D. R. Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Symposium on Operating Systems Design and Implementation*, San Diego, California, 2008. USENIX Association.
4. C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. EXE: automatically generating inputs of death. In *Conference on Computer and Communications Security*, Alexandria, Viginia, 2006. ACM.
5. L. de Moura and N. Bjørner. Z3: An Efficient SMT Solver. In *International Conference on Tools and Algorithms for the construction and Analysis of Systems*, Budapest, Hungary, 2008. Springer.

6. M. Emmi, R. Majumdar, and K. Sen. Dynamic test input generation for database applications. In *International Symposium on Software Testing and Analysis*, London, UK, 2007. ACM.
7. X. Fu, X. Lu, B. Peltsverger, S. Chen, K. Qian, and L. Tao. A static analysis framework for detecting SQL injection vulnerabilities. In *International Computer Software and Applications Conference*, Beijing, China, 2007. IEEE.
8. V. Ganesh and D. L. Dill. A decision procedure for bit-vectors and arrays. In W. Damm and H. Hermanns, editors, *19th International Conference on Computer Aided Verification (CAV 2007)*, volume 4590 of *Lecture Notes in Computer Science*, pages 519–531, Berlin, Germany, 2007. Springer.
9. P. Godefroid, A. Kiezun, and M. Y. Levin. Grammar-based whitebox fuzzing. In *Programming Language Design and Implementation*, Tuscon, Arizona, 2008. ACM.
10. P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *Programming Language Design and Implementation*, Chicago, Illinois, 2005. ACM.
11. P. Godefroid, M. Y. Levin, and D. Molnar. Automated whitebox fuzz testing. In *Network and Distributed System Security Symposium*, San Diego, California, 2008. The Internet Society.
12. S. Gulwani, S. Srivastava, and R. Venkatesan. Program analysis as constraint solving. In *Programming Language Design and Implementation*, Tuscon, Arizona, 2008. ACM.
13. W. Halfond, A. Orso, and P. Manolios. WASP: Protecting Web applications using positive tainting and syntax-aware evaluation. *Transactions on Software Engineering*, 34(1):65–81, 2008.
14. D. Jackson and M. Vaziri. Finding bugs with a constraint solver. In *International Symposium on Software Testing and Analysis*, Portland, Oregon, 2000. ACM.
15. K. Jayaraman, D. Harvison, V. Ganesh, and A. Kiezun. jFuzz: A concolic whitebox fuzzer for Java. In *NASA Formal Methods Symposium*, Moffett Field, California, 2009. NASA.
16. A. Kiezun, V. Ganesh, P. J. Guo, P. Hooimeijer, and M. D. Ernst. HAMPI: a solver for string constraints. In *International Symposium on Software Testing and Analysis*, pages 105–116, New York, NY, USA, 2009. ACM.
17. A. Kiezun, P. J. Guo, K. Jayaraman, and M. D. Ernst. Automatic creation of SQL injection and cross-site scripting attacks. In *International Conference on Software Engineering*, Vancouver, Canada, 2009. IEEE.
18. R. Majumdar and R.-G. Xu. Directed test generation using symbolic grammars. In *Automated Software Engineering"*, Atlanta, Georgia, 2007. ACM/IEEE.
19. Y. Minamide. Static approximation of dynamically generated Web pages. In *International World Wide Web Conference*, Chiba, Japan, 2005. ACM.
20. M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: engineering an efficient SAT solver. In *Design Automation Conference*, Las Vegas, Nevada, 2001. ACM.
21. D. Shannon, S. Hajra, A. Lee, D. Zhan, and S. Khurshid. Abstracting symbolic execution with string analysis. In *Testing: Academic and Industrial Conference Practice and Research Techniques*, Windsor, UK, 2007. IEEE Computer Society.
22. M. Sipser. *Introduction to the Theory of Computation*. Course Technology, Florence, KY, 2005.
23. G. Wassermann and Z. Su. Sound and precise analysis of Web applications for injection vulnerabilities. In *Programming Language Design and Implementation*, San Diego, California, 2007. ACM.
24. G. Wassermann and Z. Su. Static detection of cross-site scripting vulnerabilities. In *International Conference on Software Engineering*, Leipzig, Germany, 2008. IEEE.
25. G. Wassermann, D. Yu, A. Chander, D. Dhurjati, H. Inamura, and Z. Su. Dynamic test input generation for Web applications. In *International Symposium on Software Testing and Analysis*, Seattle, Washington, 2008. ACM.