HAMPI: A solver for string constraints

Article · July 2009

DOI: 10.1145/1572272.1572286 · Source: OAI

5 authors, including:

Adam Kiezun
Broad Institute of MIT and Harvard
100 PUBLICATIONS **5,143** CITATIONS

SEE PROFILE

Vijay Ganesh
University of Waterloo
60 PUBLICATIONS **2,802** CITATIONS

SEE PROFILE

Computer Science and Artificial Intelligence Laboratory

Technical Report

MIT-CSAIL-TR-2009-004 February 4, 2009

HAMPI: A Solver for String Constraints

Adam Kiezun, Vijay Ganesh, Philip J. Guo, Pieter Hooimeijer, and Michael D. Ernst

massachusetts institute of technology, cambridge, ma 02139 usa — www.csail.mit.edu

HAMPI: A Solver for String Constraints

Adam Kieżun
MIT
akiezun@csail.mit.edu

Vijay Ganesh
MIT
vganesh@csail.mit.edu

Philip J. Guo
Stanford University
pg@cs.stanford.edu

Pieter Hooimeijer
University of Virginia
pieter@cs.virginia.edu

Michael D. Ernst
University of Washington
mernst@cs.washington.edu

ABSTRACT

Many automatic testing, analysis, and verification 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. The increasing efficiency of off-the-shelf constraint solvers makes this approach even more compelling. However, there are few, if any, effective and sufficiently expressive off-the-shelf solvers for string constraints generated by analysis techniques for string-manipulating programs.

We designed and implemented HAMPI, a solver for string constraints over bounded string variables. HAMPI constraints express membership in regular languages and bounded context-free languages. HAMPI constraints may contain context-free-language definitions, regular-language definitions and operations, and the membership predicate. Given a set of constraints, HAMPI outputs a string that satisfies all the constraints, or reports that the constraints are unsatisfiable.

HAMPI is expressive and efficient, and can be successfully applied to testing and analysis of real programs. Our experiments use HAMPI in: static and dynamic analyses for finding SQL injection vulnerabilities in Web applications; automated bug finding in C programs using systematic testing; and compare HAMPI with another string solver. HAMPI's source code, documentation, and the experimental data are available at `http://people.csail.mit.edu/akiezun/hampi`.

1. INTRODUCTION

Many automatic testing [7, 20, 36, 42], analysis [22], and verification [9, 10] 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 the efficiency of off-the-shelf constraint solvers for Boolean SAT [12, 30] and other theories [11, 18] continues to increase. Most of these solvers are aimed at propositional logic, linear arithmetic, or the theory of bit-vectors. However, many programs, such as Web applications, take string values as input, manipulate them, and then use them in sensitive operations such as database queries. Analyses of string-manipulating programs in techniques for automatic testing [5,13,19], verifying correctness of program output [37], and finding security faults [17, 41] 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 solver that can be easily integrated into a variety of applications.

We designed and implemented HAMPI, a solver for constraints over bounded string variables. HAMPI constraints express membership in regular and bounded context-free languages[1]. HAMPI constraints may contain a bounded string variable, context-free language definitions, regular-language definitions and operations, and language-membership predicates. Given a set of constraints over a string variable, 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 regular and bounded context-free languages.

The bounding on regular and context-free languages in HAMPI's input is a key feature, different from custom string solvers used in many testing and analysis tools [13]. As we demonstrate in this paper, for many practical applications, bounding the input languages is not a handicap. In fact, it allows for a more expressive input language that allows 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 similar to the one between model-checking and bounded model-checking [4].

HAMPI's input language can encode queries such as: "Find a string v of size 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`" (`OR 1=1` is a common tautology that can lead to SQL injection attacks). HAMPI finds a string value that satisfies the constraints, or answers that no satisfying value exists (for the above example, string `1 OR 1=1` is a solution).

At a high level, HAMPI works as follows: First, HAMPI normalizes the input constraints, and generates what we refer to as the *core string constraints*. The core string constraints are expressions of the form $v \in R$ or $v \notin R$, where v is a bounded string variable, and R is a regular expression. Second, HAMPI translates these core string constraints into a quantifier-free logic of 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. Third, HAMPI hands over the bit-vector constraints to STP [18], a constraint solver for bit-vectors and arrays. Finally, if STP reports that the constraints are unsatisfiable, then HAMPI reports the same. Otherwise, STP reports

[1]All bounded languages are finite, and every finite language is regular. Hence, it would suffice to say that HAMPI supports only bounded regular languages. However, it is important to emphasize the ease-of-use that HAMPI provides by allowing users to specify context-free languages.

that the input constraints are satisfiable, and generates a satisfying assignment in its bit-vector language. HAMPI decodes this to output a string solution.

Experimental Evaluation

We experimentally evaluated HAMPI's expressiveness and efficiency over constraints obtained from the following testing and analysis applications: (i) A tool for identifying SQL injection vulnerabilities in PHP Web applications using static analysis [39], (ii) Ardilla [25], a tool for creating SQL injection attacks using dynamic analysis of PHP Web applications, (iii) Klee [6], a systematic testing tool. Additionally (iv), we compared HAMPI's performance to CFGAnalyzer [1], a solver for analyzing context-free grammars. The experimental results indicate that HAMPI is efficient, and its input language can express string constraints that arise from a variety of real-world analysis and testing tools.

Summary of Results

- **SQL Injection Vulnerability Detection:** We applied the static analysis tool [39] 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 on real applications bolster our claim that bounding the string constraints is not a handicap.
- **SQL Injection Attack Generation:** We applied Ardilla [25] to 5 PHP Web applications (total lines of code: 14,941). HAMPI successfully replaced a custom-made attack generator and constructed all 23 known attacks on those applications.
- **Input Generation for Systematic Testing:** We used Klee [6] on 3 C 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× (up to 5× in parser code), and discovered 3 new bug-revealing inputs.
- **Comparison with CFGAnalyzer:** HAMPI is, on average, 6.8 times faster than CFGAnalyzer [1] on 100 grammar intersection problems. Furthermore, HAMPI's speedup increased with the problem size.

Contributions

- A *decision procedure* for constraints over bounded string variables, supporting regular language membership, context-free language membership, and typical string operations like concatenation.
- HAMPI, an open-source *implementation* of the decision procedure. HAMPI's source code and documentation are available at: `http://people.csail.mit.edu/akiezun/hampi`.
- Experimental *evaluation* of HAMPI for a variety of testing and analysis applications.
- Downloadable (from HAMPI website) *experimental data* that can be used as benchmarks for developing and evaluating future string solvers.

We introduce HAMPI using an example (§2), then present HAMPI's input format and solving algorithm (§3), discuss speed optimizations (§4), and present the experimental evaluation (§5). We finish with related work (§6) and conclusion (§7).

```
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 }
```

Figure 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 '$my_topicid' from Figure 1
2  var v:12;     // size is 12 characters
3
4  //simple SQL context-free grammar
5  cfg SqlSmall := "SELECT " [a-z]+ " FROM " [a-z]+ " WHERE " Cond;
6  cfg Cond     := Val "=" Val | Cond " OR " Cond";
7  cfg Val      := "'" [a-z0-9]* "'" | [0-9]+;
8
9  //SQL grammar bounded to 53 characters
10 reg SqlSmallBounded := bound(SqlSmall, 53);
11
12 //the SQL query '$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 SqlSmallBounded;
17 assert q contains "OR '1'='1'";
```

Figure 2: The HAMPI input that finds an attack vector that exploits the SQL injection vulnerability from Figure 1.

2. EXAMPLE: SQL INJECTION

SQL injections are a prevalent class of Web-application vulnerabilities. This section illustrates how an automated tool [25, 41] could use HAMPI to detect SQL injection vulnerabilities and to produce attack inputs.

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 from the database (including ones intended to be private) by crafting a malicious URL such as

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

Upon being invoked with that URL, the program reads

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

as the value of the `$my_topicid` variable, 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 a bug-finding tool might ask, "Can an attacker exploit the `topicid` parameter and introduce a tautology into the 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.

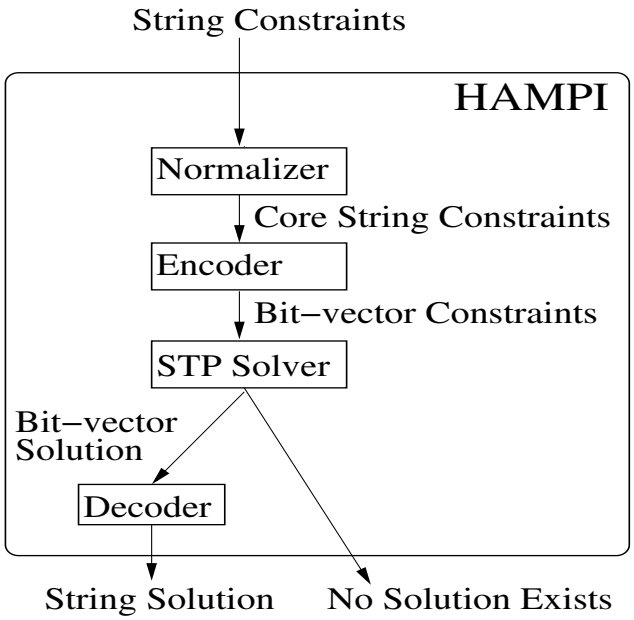

Figure 3: Schematic view of the HAMPI string solver. Section 3 describes the HAMPI solver.

Input	::=	*Var Stmt*∗	HAMPI input
Stmt	::=	*Cfg* | *Reg* | *Val* | *Assert*	statement
Var	::=	**var** *Id* : *Int*	string variable
Cfg	::=	**cfg** *Id* := *CfgProdRHS*	context-free lang.
Reg	::=	**reg** *Id* := *RegElem*	regular-lang.
RegElem	::=	*StrConst*	constant
	|	*Id*	var. reference
	|	**bound**(*Id* , *Int*)	CFG bounding
	|	**or**(*RegElem* ∗)	union
	|	**concat**(*RegElem* ∗)	concatenation
	|	**star**(*RegElem*)	Kleene star
Val	::=	**val** *Id* :=*ValElem*	temp. variable
ValElem	::=	*Id* | *StrConst* | **concat**(*ValElem* ∗)	
Assert	::=	**assert** *Id* [**not**]? **in** *Id*	membership
	|	**assert** *Id* [**not**]? **contains** *StrConst*	substring

Figure 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 statements: context-free-grammar declarations, regular-language declarations, temporary variables, and assertions. Some *nonterminals* are omitted for readability; the HAMPI website contains the formal grammar and documentation of the input format.

Figure 2 presents a HAMPI constraint that formalizes the above question. Automated vulnerability-scanning tools [25, 41] can create such constraints via either static or dynamic program analysis (we demonstrate both static and dynamic techniques in our evaluation in Sections 5.1 and 5.2). Specifically, a tool could create the HAMPI input of Figure 2 from 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 fixed size of 12 characters. The goal of the HAMPI solver is to find a string that, when assigned to the string variable, satisfies all the constraints. HAMPI can look for solutions of any fixed size; we chose 12 for this example.
- Keyword `cfg` (lines 5–7) introduces a *context-free grammar*, for a fragment of the SQL grammar of `SELECT` statements.
- Keyword `reg` (line 10) introduces a regular expression `SqlSmallBounded`, defined by *bounding* the context-free grammar (of strings derivable from `SqlSmall`) to a fixed size of 53 characters. The size is chosen to be consistent with the size of `q`, which is the sum of the size of `v` (12) and the sizes of the constant strings (40+1) in the expression that defines `q` (line 13).
- 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 regular-language 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 regular language `SqlSmallBounded`. Line 17 specifies that the variable `v` must contain a specific substring (e.g., a 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 `topicid` that can lead to an SQL injection attack. This value has exactly 12 characters, since `v` was defined with that fixed size. By re-running HAMPI with different sizes for `v`, it's possible to create other (usually related) attack inputs, such as `999' OR '1'='1`.

3. THE HAMPI STRING SOLVER

The HAMPI solver finds a string that satisfies constraints specified in the input, or finds that no satisfying string exists. Figure 3 shows HAMPI's architecture. HAMPI works in four steps:

1. Normalize the input constraints to a *core form* (Section 3.2).
2. Encode the constraints in bit-vector logic (Section 3.3).
3. Invoke the STP bit-vector solver [18].
4. Decode the results obtained from STP (Section 3.3).

3.1 Input Language for String Constraints

We discuss the salient features of HAMPI's input language (Figure 4) and illustrate on examples. HAMPI's input language enables encoding of string constraints generated from typical testing and security applications. The language supports declaration of bounded string variables and constants, regular-language operations, membership predicate, and declaration of context-free and regular languages, temporaries and constraints.

String variable declaration — `var`

A HAMPI input must declare a *single* string variable and specify the variable's size in number of characters. If the input constraints are satisfiable, then HAMPI finds a value for the variable that satisfies all constraints. Line 2 in Figure 2 declares a variable `v` of size 12 characters.

Sometimes, an application of a string solver requires examining strings *up to* a given length. Users of HAMPI can deal with this

issue in two ways: (i) repeatedly run HAMPI for different fixed sizes of the variable (can be fast due to the optimizations of Section 4), or (ii) adjust the constraint to allow "padding" of the variable (e.g., using Kleene star to denote trailing spaces). We are considering extending HAMPI to permit specifying a size range, using syntax such as `var v:1..12`.

Declarations of Context-free Languages — `cfg`

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 5–7 in Figure 2 show the declaration of a context-free grammar for a subset of SQL.

HAMPI's format of 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).

Declarations of Regular Languages — `reg`

HAMPI input can declare regular languages. The following regular expressions define regular languages: (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) a fixed-size "bounding" of a context-free language. Every regular language can be expressed using those operations [38].

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`'s. In HAMPI syntax this is:

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

HAMPI allows construction of regular languages by bounding context free languages. The set of all strings of a given size from a context-free language is regular (because every finite language is regular). In HAMPI, only regular languages can be used in constraints. Therefore, every context-free grammar must be bounded before being used in a constraint.

For example, in line 10 of Figure 2, the regular language described by `SqlSmallBounded` consists of all syntactically correct SQL strings of length 53 (according to the `SqlSmall` grammar). Using the **bound** operator is much more convenient than writing the regular expression explicitly.

Temporary Declarations — `val`

Temporary variables are shortcuts for expressing constraints on expressions that are concatenations of the string variable and constants.

Line 13 in Figure 2 declares a temporary variable `val q` that denotes the SQL query, which is a concatenation of two string constants (prefix and suffix) and the string variable `v`. Using `q` is a convenient shortcut to put constraints on that SQL query (lines 16 and 17).

Constraints — `assert`

HAMPI constraints (declared by the `assert` keyword) specify membership of variables in regular languages. For example, line 16 in Figure 2 declares that the string value of the temporary variable `q` is in the regular language defined by `SqlSmallBounded`.

S	::=	*Constraint*	
	|	*S* ∧ *Constraint*	conjunction
Constraint	::=	*StrExp* ∈ *RegExp*	membership
	|	*StrExp* ∉ *RegExp*	non-membership
StrExp	::=	*Var*	variable
	|	*Const*	constant
	|	*StrExp StrExp*	concatenation
RegExp	::=	*Const*	constant
	|	*RegExp* + *RegExp*	union
	|	*RegExp RegExp*	concatenation
	|	*RegExp*⋆	star

Figure 5: The grammar of core string constraints. Nonterminals *Const* and *Var* have the usual definitions.

3.2 Core Form of String Constraints

After parsing and checking the input, HAMPI normalizes the string constraints to a core form (Figure 5). The core string constraints are an internal intermediate representation that is easier to encode in bit-vector logic than raw HAMPI input is.

A core string constraint specifies membership (or its negation) in a regular language. A core string constraint is in the form *StrExp* ∈ *RegExp* or *StrExp* ∉ *RegExp*, where *StrExp* is an expression composed of concatenations of string constants and occurrences of the string variable, and *RegExp* is a regular expression.

HAMPI normalizes each component in the input as follows:

1. Expand all temporary variables, i.e., replace each reference to a temporary variable with the variable's definition.
2. Expand all context-free grammar bounding expressions, i.e., convert **bound** terms to 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 [38].
3. Construct the regular expression that encodes all fixed-sized strings of the grammar as follows: First, pre-process the grammar and find the length of the shortest and longest (if exists) string that can be generated from each nonterminal. 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, create the sub-expressions recursively and combine the subexpression in with a concatenation operator. Memoization of intermediate results (Section 4.1) makes this (worst-case exponential in k) process scalable.

Example of Grammar Bounding. 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`.

Formula	::=	*BitVector* = *BitVector*	equality
	|	*BitVector* < *BitVector*	inequality
	|	*Formula* ∨ *Formula*	disjunction
	|	*Formula* ∧ *Formula*	conjunction
	|	¬*Formula*	negation
BitVector	::=	*Const*	bit-vector constant
	|	*Var*	bit-vector variable
	|	*Var*[*Int*]	byte extraction

Figure 6: Grammar of bit-vector logic. Variables denote bit-vectors of fixed length. HAMPI encodes string constraints as formulas in this logic and solves using STP.

```
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 string constraints as formulas in a logic of fixed-size bit-vectors. A bit-vectors 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 (Figure 6). HAMPI asks STP 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 string constraint is encoded separately, as a conjunct in a bit-vector logic formula. HAMPI encodes the core string constraint *StrExp* ∈ *RegExp* recursively, by case analysis of the regular expression *RegExp*, as follows:

- Encoding constants — the encoded formula enforces 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 Complexity

The satisfiability problem for HAMPI's logic (core string constraints) is NP-complete. To show NP-hardness, we reduce the 3-CNF (conjunctive normal form) Boolean satisfiability problem to the satisfiability problem of the core string constraints in HAMPI's logic. Consider an arbitrary 3-CNF formula with n Boolean variables and m clauses. A clause in 3-CNF is a disjunction ($\vee$) of three literals. A literal is a Boolean variable (v_i) or its negation ($\neg v_i$). For every 3-CNF clause, a HAMPI constraint can be generated. Let Σ denote the alphabet {T, F}. For the clause ($v_0 \vee v_1 \vee \neg v_2$), the equivalent HAMPI constraint is:

$$V \in (\mathrm{T}\Sigma\Sigma\cdots\Sigma \;+\; \Sigma\mathrm{T}\Sigma\cdots\Sigma \;+\; \Sigma\Sigma\mathrm{F}\cdots\Sigma)$$

where the HAMPI variable V is an n character string representing the possible assignments to all n Boolean variables satisfying the input 3-CNF formula. Each of the HAMPI regular expression disjuncts in the core string constraint shown above, such as $T\Sigma\Sigma\cdots\Sigma$, is also of size n and has a T in the i^{th} slot for v_i (and F for $\neg v_i$). The total number of such HAMPI constraints is m, the number of clauses in the input 3-CNF formula (here $m = 1$). This reduction from a 3-CNF Boolean formula into HAMPI is clearly polynomial-time.

To establish that the satisfiability problem for HAMPI's logic is in NP, we only need to show that for any set of core string constraints, there exists a polynomial-time verifier that can check any short witness. The size of a set of core string constraints is the size k of the string variable plus the sum r of the sizes of regular expressions. A witness has to be of size k, and it is easy to check if the witness belongs to each regular language in time polynomial in $k + r$.

3.5 Example of Solving

This section illustrates how, given the following input, HAMPI finds a satisfying assignment for variable v.

```
var v:2;
cfg E := "()" | E E | "(" E ")";
reg Ebounded := bound(E, 6);
val q := concat( "((" , v , "))" );
assert q in Ebounded;    // turns into constraint c1
assert q contains "())"; // turns into constraint c2
```

HAMPI follows the solving algorithm outlined in Section 3 (The alphabet of the regular expression or context-free grammar in a HAMPI input is implicitly restricted to the terminals specified):

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

```
c1:  ((v))  ∈  ()[()() + (())] +
               [()() + (())]() +
               ([()() + (())])
c2:  ((v))  ∈  [( + )]⋆ () [( + )]⋆
```

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 `Ebounded` 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:

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

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 is a 3-way disjunction. For the first disjunct `()[()() + (())]`, Hampi creates the following formula: D_{1a}: $bv[0] = 40 \wedge bv[1] = 41 \wedge \big((bv[2] = 40 \wedge bv[3] = 41 \wedge bv[4] = 40 \wedge bv[5] = 41\big) \vee \big(bv[2] = 40 \wedge bv[3] = 40 \wedge bv[4] = 41 \wedge bv[5] = 41\big)\big)$.

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 $(C_1 \wedge C_2)$.

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).

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. It reports the solution for v as "`)(`". (String "`()`" is another legal solution for v, but STP only finds one solution.)

4. OPTIMIZATIONS

We implemented several optimizations in Hampi, aimed at reducing computation time.

4.1 Memoization

Hampi stores and reuses partial results during the computation of bounding of context-free grammars (Section 3.2) and during the encoding of core constraints in bit-vector logic (Section 3.3).

Example. Consider the example from Section 3.5, i.e., bounding the context-free grammar of well-balanced parentheses to size 6.

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

Consider the second production `E := E E`. There are two ways to construct a string of 6 characters: either construct 2 characters from the first occurrence of `E` and construct 4 characters from the second occurrence, or vice-versa. After creating the regular expression that corresponds to the first of these ways, Hampi creates the second expression from the memoized sub-results. Hampi's implementation shares the memory representations of common subexpressions. For example, Hampi uses only one object to represent all three occurrences of `()() + (())` in constraint **c1** of the example in Section 3.5.

4.2 Constraint Templates

During the bit-vector encoding step (Section 3.3), Hampi may encode the same regular expression multiple times as bit-vector formulas, as long as the underlying offsets in the bit-vector are different. For example, the (constant) regular expression `)(` may be encoded as $bv[0] = 41 \wedge bv[1] = 40$ or as $bv[3] = 41 \wedge bv[4] = 40$, depending on the offset in the bit-vector (0 and 3, respectively).

As an optimization, Hampi creates a single "template", parameterized by the offset, for the encoded expression, and instantiates the template every time, with appropriate offsets. For the example above, the template $T(p)$ is $bv[p] = 41 \wedge bv[p+1] = 40$, where p is the offset parameter. Hampi then instantiates the template to $T(0)$ and $T(3)$.

As another example, consider **c1** in Section 3.5: The subexpression `()()+(())` occurs 3 times in **c1**, each time with a different offset (2 for the first occurrence, 0 for the second, and 1 for the third). The constraint template optimization enables Hampi to do the encoding once and reuse the results, with appropriate offsets.

4.3 Server Mode

Because Hampi is a Java program, the startup time of the Java virtual machine may be a significant overhead when solving small constraints. Therefore, we added a server mode to Hampi, in which the (constantly running) solver accepts inputs passed over a network socket, and returns the results over the same socket. This enables Hampi to be efficient over repeated calls, for tasks such as solving the same constraints on string variables of different sizes.

5. EVALUATION

We performed experiments to test Hampi's applicability to practical problems involving string constraints, and to compare its performance and scalability to another string solver.

Experiments:

1. We used Hampi in a static-analysis tool [39] that identifies possible SQL injection vulnerabilities (Section 5.1).

2. We uses Hampi in Ardilla [25], a dynamic-analysis tool that creates SQL injection attacks (Section 5.2).

3. We used Hampi in Klee, a systematic testing tool for C programs (Section 5.3).

4. We compared Hampi's performance and scalability to CFG-Analyzer [1], a solver for bounded versions of context-free-language problems, e.g., intersection (Section 5.4).

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 indicate 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`.

5.1 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 [39] 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 would offer two important advantages. First, it would eliminate 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 [29]. 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.

However, an advantage of using an unbounded string solver is that if the solver terminates and says that the input constraints have no solution, then there is indeed no solution. In the case of Hampi, on the other hand, we can only conclude that there is no solution up to the given bound.

We performed the experiment as follows: We applied Wassermann and Su's tool to 6 PHP applications. Of these, 5 were applications used by Wassermann and Su to evaluate their tool [39]. We added 1 large application (`claroline 1.5.3`, a builder for online education courses, with 169 KLOC) from another paper by the same authors [40]. 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.

In 306 of the 1,367 inputs, the intersection was *not* empty (both solvers produced identical results). 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.

We found empirically that when a solution exists, it can be very short. 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 $N \geq 4$. 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. Figure 7 shows the solving time as a function of the grammar size, for all 1,367 inputs.

Hampi can solve most queries quickly. Figure 8 shows the percentage of inputs that Hampi can solve in the given time, for $1 \leq N \leq 4$, i.e., until all satisfying assignments are found. For $N = 4$, Hampi can solve 99.7% of inputs within 1 second.

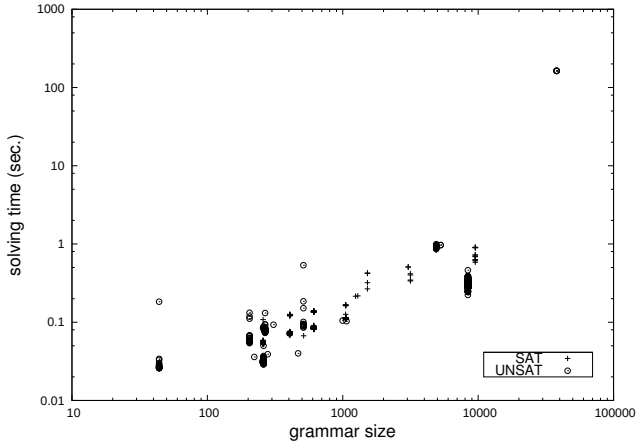

Figure 7: Hampi solving time as function of grammar size (number of all elements in all productions), on 1,367 inputs from the Wasserman and Su dataset [39]. The size of the string variable was 4, the smallest at which Hampi finds all satisfying assignments for the dataset. Each point represents an input; shapes indicate SAT/UNSAT. Section 5.1 describes the experiment.

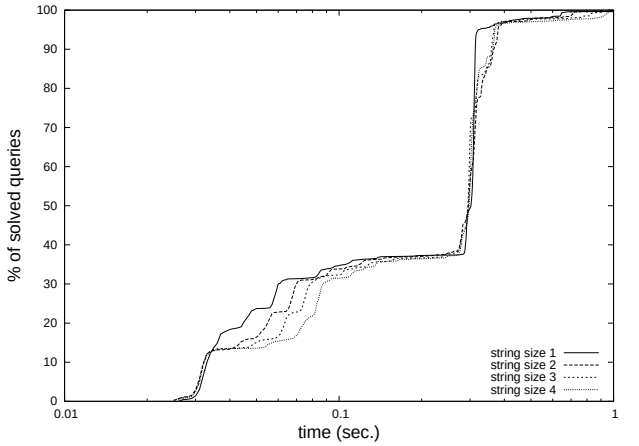

Figure 8: Percentage of queries solvable by Hampi, in a given amount of time, on data from Wasserman and Su [39]. Each line represents a distribution for a different size of the string variable. All lines reach 99.7% at 1 second and 100% before 160 seconds. Section 5.1 describes the experiment.

Summary of results: We applied Hampi to 1,367 constraints created from analysis of 339,750 lines of code from 6 PHP applications. Hampi found that all 306 satisfiable constraints have short solutions ($N \leq 4$). Hampi found all known solutions, and solved 99.7% of the generated constraints in less than 1 second per constraint. These result, obtained on a large dataset from a powerful static analysis and real Web applications, indicates that Hampi's bounded solving algorithm is applicable to real problems.

5.2 Creating SQL Injection Attacks from 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 [25], 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 [20, 36] component. This required code changes were quite extensive 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 manually 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 14 nonterminals, 16 terminals, and 27 productions (its size is 74, according to the metric of Section 5.1). 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 [25], 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 [17, 23], 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[2].

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.

Summary of results: HAMPI fully replaced Ardilla's custom attack generator. HAMPI successfully created all 23 attacks on the tested applications. HAMPI solved the associated constraints quickly, finding all known solutions for $N \leq 6$. HAMPI solved 46.0% of those constraints in less than 1 second per constraint, and solved all the constraints in less than 10 seconds per constraint.

These results indicate that the HAMPI enabled a successful reduction of the problem of finding SQL injection attacks to string constraint generation and solving, and was able to plug into an existing security testing application and perform comparably.

[2] `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`

5.3 Systematic Testing of C Programs

We combined HAMPI with a state-of-the-art systematic testing tool, Klee [6], to improve Klee's ability to create valid test cases for C programs that accept highly structured string inputs.

Automatic test-case generation tools that use combined concrete and symbolic execution, also known as called *concolic execution* (e.g., CUTE [36], DART [20], EXE [7], Klee [6], SAGE [21]) 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 [19, 27] 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 [6] to generate test cases for C 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 [19,27], and we do not make any claims of novelty for the experimental design. However, previous studies used custom-made string solvers, while we are able to apply HAMPI as an "off-the-shelf" solver for Klee without modifying Klee at all.

Klee provides a C 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 numeric 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
```

Such simple constraints 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 C programs that specify expected inputs using context-free grammars in Yacc format (a subset of those used by Majumdar and Xu [27]). `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 3.2GHz Pentium 4 PC with 1GB of RAM running Fedora Linux:

Program	ELOC	input size		symbolic	symbolic + grammar	combined
`cueconvert`	939	28 bytes	% 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	7 bytes	% 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	6 bytes	% 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 5.3 describes the experiment.

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 bounded-length restriction to limit the input to N bytes. Klee (similarly to, for example, SAGE [21]) actually requires a fixed-length input, which matches well with HAMPI's bounded 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.

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, bounding to N bytes, running HAMPI on the bounded grammar, and converting the resulting STP constraints into C code are negligible: They took less than 1 second for each of the 3 programs.

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.

Using the grammar, Klee generated 3 distinct inputs for `logictree` that uncovered (previously unknown) bugs where the program entered an infinite loop. 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.

Upon manually inspecting lines of code that were not covered (even in the combined column), we discovered two main hindrances to achieving higher coverage: First, our 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).

Summary of results: Using HAMPI to create input constraints led to up to 2× improvements in line coverage (up to 5× coverage improvements in parser code), eliminated all illegal inputs, and enabled discovering 3 distinct, previously unknown, inputs that led to infinitely-looping program execution. These results show that using HAMPI can improve the effectiveness of automated test-case generation and bug finding tools.

5.4 Comparing Performance to CFGAnalyzer

We evaluated HAMPI's utility in analyzing context-free grammars, and compared HAMPI's performance to a specialized decision pro-

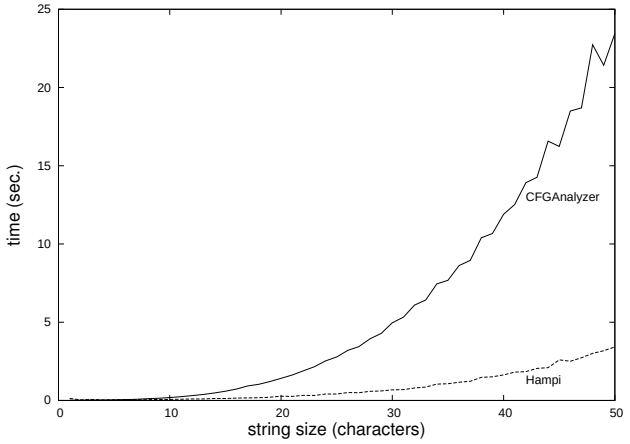

Figure 9: Solving time as a function of string size, on context-free-grammar intersection constraints. Results are averaged over 100 randomly-selected pairs of context-free grammars. Section 5.4 describes the experiment.

cedure, CFGAnalyzer [1]. CFGAnalyzer is a SAT-based decision procedure for bounded versions of 6 problems (5 undecidable) that involve context-free grammars: universality, inclusion, intersection, equivalence, ambiguity, and emptiness (decidable). We downloaded the latest available version[3] (released 3 December 2007) and configured the program according to the manual. To avoid bias, we compared HAMPI to CFGAnalyzer only on the data provided by CFGAnalyzer developers.

We performed experiments with the grammar intersection problem. Five of six problems handled by CFGAnalyzer (universality, inclusion, intersection, equivalence, and emptiness) can be easily encoded as HAMPI inputs — the intersection problem is representative of the rest. In these experiments, both HAMPI and CFGAnalyzer searched for strings (of fixed length) from the intersection of 2 grammars. We used CFGAnalyzer's experimental data sets (obtained from the authors). From the set of 2088 grammars in the data set, we selected a random sample of 100 grammar pairs. We used both HAMPI and CFGAnalyzer to search for strings of lengths $1 \leq N \leq 50$. We ran CFGAnalyzer in a non-incremental mode (in the incremental mode, CFGAnalyzer reuses previously computed sub-solutions), to create a fair comparison with HAMPI, which ran as usual in server mode while flushing its entire internal state after solving each input. We ran both programs without a timeout.

Figure 9 shows the results averaged over all pairs of grammars. HAMPI is faster than CFGAnalyzer for all sizes larger than 4 characters. More importantly, HAMPI's win over CFGAnalyzer grows as the size of the problem increases (up to 6.8× at size 50). For the largest problems ($N = 50$), HAMPI was faster (by up to 3000×) on 99 of the 100 grammar pairs, and 1.3× slower on the remaining 1 pair of grammars.

The HAMPI website contains all experimental data and detailed results. It also describes an additional experiment we performed: searching for any string of a given length from a context-free grammar. The results were similar to those for intersection: e.g., HAMPI finds a string of size 50, on average, in 1.5 seconds, while CFGAnalyzer finds one in 8.7 seconds (5.8× difference).

Summary of results: On average, HAMPI solved constraints up to 6.8× faster than CFGAnalyzer, and its lead increased as the problem size grew larger.

[3] `http://www.tcs.ifi.lmu.de/~mlange/cfganalyzer`

6. RELATED WORK

Decision procedures have received widespread attention within the context of program analysis, testing, and verification. There has been significant work on decision procedures for theories such as Boolean satisfiability [12, 30], bit-vectors [18], quantified Boolean formulas [2, 3], and linear arithmetic [11]. In contrast, there has been relatively little work on practical and expressive solvers that reason about strings or sets of strings directly.

Solvers for String Constraints. MONA [26] uses finite-state automata and tree automata to reason about sets of strings. However, the user still has to translate their input problem into MONA's input language (weak monadic second-order theory of one successor). MONA also provides automata-based tools, similar to other libraries [14–16].

In concurrent work, Hooimeijer and Weimer [24] describe a decision procedure for regular language constraints, focusing on generating sets of satisfying assignments rather than individual strings. Unlike HAMPI, the associated implementation does not easily allow users to express bounded context-free grammars.

Several tools allow word equations [5, 33], i.e., checking equality between two strings that contain string variables. Rajasekar [33] exhibits a logic programming approach that includes constraints on individual words. Bjørner et al. [5] describe a constraint solver for word queries over a variety of operations, and translate string constraints to the language of the Z3 solver [11]. If there is a solution, Z3 returns a finite bound for the set of strings, that is then explored symbolically. However, unlike HAMPI, these tools do not support context-free grammars directly.

Custom String Solvers. Many analyses use custom solvers to solve string constraints [8, 13, 17, 19, 29, 39–41]. All of these approaches include some implementation for language intersection and language inclusion; most, like HAMPI, can perform regular language intersection. Each of these implementations is tightly integrated with the associated program analysis, making a direct comparison with HAMPI impractical.

Christensen et al. [8] have a static analysis tool to check for SQL injection vulnerabilities that uses automata-based techniques to represent over-approximation of string values. Fu et al. [17] also use an automata-based method to solve string constraints. Ruan et al [34] use a first-order encoding of string functions occurring in C programs, and solve the constraints using a linear arithmetic solver.

Besides the custom solvers by Wasserman et al. [39], the solver by Emmi et al. [13] is closest to HAMPI in terms of expressiveness and usage for testing and analysis applications. Emmi et al. used their solver for automatic test case generation for database applications. Unlike HAMPI, their solver allows constraints over unbounded regular languages and linear arithmetic, but does not support context-free grammars.

Many of the program analyses listed here perform similar tasks when reasoning about string-valued variables. This is strong evidence that a unified approach, in the form of an external string constraint solvers like HAMPI, is warranted.

Theoretical Work on Satisfiability of String Constraints: There are a variety of inter-related problems involving strings constraints, and there is an extensive literature on the theoretical study of these problems [28, 31, 32, 35]. However, we are focused on efficient techniques for a practical string solver that is usable as a library and is sufficiently expressible to support a large variety of applications.

7. CONCLUSION

We presented HAMPI, a solver for constraints over bounded string variables. HAMPI constraints express membership in regular and bounded context-free languages. HAMPI constraints may contain a bounded string variable, context-free language definitions, regular-language definitions and operations, and language-membership predicates. Given a set of constraints over a string variable, HAMPI outputs a string that satisfies all the constraints, or reports that the constraints are unsatisfiable. HAMPI works by encoding the constraint in the bit-vector logic and solving using STP.

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 regular and bounded context-free languages. We evaluated HAMPI's usability and effectiveness as a component in static and dynamic analysis tools for PHP Web applications. Our experiments show that HAMPI is expressive enough to easily encode constraint arising in finding SQL injection attacks, and in systematic testing of real-world programs. In our experiments, HAMPI was able to find solutions quickly, and scale to practically-relevant problem sizes.

By using a general-purpose freely-available string solver like HAMPI, builders of analysis and testing tools can save significant development effort, and improve the effectiveness of their tools.

8. REFERENCES

[1] R. Axelsson, K. Heljank, and M. Lange. Analyzing context-free grammars using an incremental SAT solver. In *ICALP*, 2008.

[2] M. Benedetti. sKizzo: a suite to evaluate and certify QBFs. In *CADE*, 2005.

[3] A. Biere. Resolve and expand. In *SAT*, 2005.

[4] A. Biere, A. Cimatti, E. Clarke, O. Strichman, and Y. Zhu. Bounded model checking. *Advances in Computers*, 58, 2003.

[5] N. Bjørner, N. Tillmann, and A. Voronkov. Path feasibility analysis for string-manipulating programs. Technical Report MSR-TR-2008-153, Microsoft, 2008.

[6] C. Cadar, D. Dunbar, and D. R. Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, 2008.

[7] C. Cadar, V. Ganesh, P. M. Pawlowski, D. Dill, and D. R. Engler. EXE: automatically generating inputs of death. In *CCS*, 2006.

[8] A. S. Christensen, A. Møller, and M. I. Schwartzbach. Precise analysis of string expressions. In *SAS*, 2003.

[9] E. M. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In *TACAS*, 2004.

[10] E. M. Clarke, D. Kroening, N. Sharygina, and K. Yorav. Predicate abstraction of ANSI-C programs using SAT. *Formal Methods in System Design*, 25(2-3), 2004.

[11] L. DE Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS*, 2008.

[12] N. Eén and N. Sörensson. An extensible SAT solver. In *SAT*, 2003.

[13] M. Emmi, R. Majumdar, and K. Sen. Dynamic test input generation for database applications. In *ISSTA*, 2007.

[14] Brics finite state automata utilities. http://www.brics.dk/automaton/faq.html.

[15] Finite state automata utilities. http://www.let.rug.nl/ vannoord/Fsa/fsa.html.

[16] AT&T Finite State Machine Library. http://www.research.att.com/ fsmtools/fsm.

[17] X. Fu, X. Lu, B. Peltsverger, S. Chen, K. Qian, and L. Tao. A static analysis framework for detecting SQL injection vulnerabilities. In *COMPSAC*, 2007.

[18] V. Ganesh and D. L. Dill. A decision procedure for bit-vectors and arrays. In *CAV*, 2007.

[19] P. Godefroid, A. Kieżun, and M. Y. Levin. Grammar-based whitebox fuzzing. In *PLDI*, 2008.

[20] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *PLDI*, 2005.

[21] P. Godefroid, M. Y. Levin, and D. Molnar. Automated whitebox fuzz testing. In *NDSS*, 2008.

[22] S. Gulwani, S. Srivastava, and R. Venkatesan. Program analysis as constraint solving. In *PLDI*, 2008.

[23] W. Halfond, A. Orso, and P. Manolios. WASP: Protecting Web applications using positive tainting and syntax-aware evaluation. *IEEE TSE*, 34(1):65, 2008.

[24] P. Hooimeijer and W. Weimer. A decision procedure for subset constraints over regular languages. In *PLDI*, 2009.

[25] A. Kieżun, P. J. Guo, K. Jayaraman, and M. D. Ernst. Automatic creation of SQL injection and cross-site scripting attacks. In *ICSE*, 2009.

[26] N. Klarlund. Mona & Fido: The logic-automaton connection in practice. In *International Workshop on Computer Science Logic*, 1998.

[27] R. Majumdar and R.-G. Xu. Directed test generation using symbolic grammars. In *ASE*, 2007.

[28] G. Makanin. The problem of solvability of equations in a free semigroup. *Sbornik: Mathematics*, 32(2):129–198, 1977.

[29] Y. Minamide. Static approximation of dynamically generated Web pages. In *WWW*, 2005.

[30] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: engineering an efficient SAT solver. In *DAC*, 2001.

[31] G. Pesant. A regular language membership constraint for finite sequences of variables. In *CP*, 2004.

[32] C. Quimper and T. Walsh. Global grammar constraints. In *CP*, 2006.

[33] A. Rajasekar. Applications in constraint logic programming with strings. In *PPCP*, 1994.

[34] H. Ruan, J. Zhang, and J. Yan. Test data generation for C programs with string-handling functions. In *TASE*, 2008.

[35] M. Sellmann. The theory of grammar constraints. In *CP*, 2006.

[36] K. Sen, D. Marinov, and G. Agha. CUTE: A concolic unit testing engine for C. In *FSE*, 2005.

[37] D. Shannon, S. Hajra, A. Lee, D. Zhan, and S. Khurshid. Abstracting symbolic execution with string analysis. In *TAICPART*, 2007.

[38] M. Sipser. *Introduction to the Theory of Computation*. Course Technology, 1996.

[39] G. Wassermann and Z. Su. Sound and precise analysis of Web applications for injection vulnerabilities. In *PLDI*, 2007.

[40] G. Wassermann and Z. Su. Static detection of cross-site scripting vulnerabilities. In *ICSE*, 2008.

[41] G. Wassermann, D. Yu, A. Chander, D. Dhurjati, H. Inamura, and Z. Su. Dynamic test input generation for Web applications. In *ISSTA*, 2008.

[42] Y. Xie and A. Aiken. Saturn: A scalable framework for error detection using Boolean satisfiability. In *CAV*, 2007.