Effective Software Testing with a String-Constraint Solver

by

Adam Kieżun

M.Sc., University of Warsaw, Poland
B.Sc., University of Warsaw, Poland

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2009

Author . . . .
Department of Electrical Engineering and Computer Science
May 21, 2009

Certified by . . . .
Michael D. Ernst
Associate Professor
Thesis Supervisor

Accepted by . . . .
Terry P. Orlando
Chairman, Department Committee on Graduate Students

Effective Software Testing with a String-Constraint Solver

by

Adam Kieżun

Submitted to the Department of Electrical Engineering and Computer Science
on May 21, 2009, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Abstract

This dissertation presents techniques and tools for improving software reliability, by using an expressive string-constraint solver to make implementation-based testing more effective and more applicable. Concolic testing is a paradigm of implementation-based systematic software testing that combines dynamic symbolic execution with constraint-based systematic execution-path enumeration. Concolic testing is easy to use and effective in finding real errors. It is, however, limited by the expressiveness of the underlying constraint solver. Therefore, to date, concolic testing has not been successfully applied to programs with highly-structured inputs (e.g., compilers), or to Web applications.

This dissertation shows that the effectiveness and applicability of concolic testing can be greatly improved by using an expressive and efficient *string-constraint solver*, i.e., a solver for constraints on string variables. We present the design, implementation, and experimental evaluation of a novel string-constraint solver. Furthermore, we show novel techniques for two important problems in concolic testing: getting past input validation in programs with highly-structured inputs, and creating inputs that demonstrate security vulnerabilities in Web applications.

Thesis Supervisor: Michael D. Ernst
Title: Associate Professor

Acknowledgments

The years I spent working on my PhD degree at MIT have undoubtedly been some of the most joyful and rewarding in my life. They provided me with everything I need to thrive: challenging research problems, excellent company, and a supportive environment. I am deeply grateful to the people who shared this experience with me and to those who made it possible.

Special thanks go to my advisor, prof. Michael Ernst. Mike has provided me with the invaluable support and the independence I needed. He has helped me develop as a better researcher, critical thinker, writer, and presenter. His insistence on excellence has influenced me profoundly and helped me aspire to become great at everything I do. I also thank my committee members, prof. Daniel Jackson and prof. Srini Devadas, for their insights and support.

I am very grateful to Frank Tip from IBM Research. He has been a tireless supporter of me before, during, and after my stay at MIT. Frank's integrity has been an inspiration to me and I see him as an example to follow on how to approach research problems and collaboration.

I have had many great colleagues at MIT: Danny Dig, Alan Donovan, Vijay Ganesh, Philip Guo, Sung Kim, Stephen McCammant, Carlos Pacheco, Jeff Perkins, Derek Rayside, Yoav Zibin. Their supportive companionship has been very important to me, and I learned something unique and valuable from each of them.

Shay Artzi, my office mate, deserves a special thank-you. We have collaborated on more research projects than I can remember, and working with him has always been rewarding and fun. He has been an excellent friend and I admire his efficiency, sense of humor, and intelligence.

Noam Shomron has been instrumental in introducing me to the fascinating field of bioinformatics and in supporting my development and career. His patience in discussing biology with me, his incredible enthusiasm for collaborative research, and his big-picture approach to science have been an inspiration to me.

I thank my colleagues and mentors outside of MIT. Erich Gamma, Dirk Bäumer, Steve Northover, Kevin Haaland, and others from IBM/OTI have provided me with a great deal of support, technical expertise, and a pragmatic perspective on the world. I learned a lot from my collaborators during my summer internships: Michael Levin and Patrice Godefroid at Microsoft, as well as Frank Tip and Robert Fuhrer at IBM Research.

My final thanks goes to my family. I am indebted to my parents for the fantastic love, support, and opportunities they have given me. Gosia, my lovely wife, has provided me with her boundless love and support. My stay at MIT has been a great adventure and there is no one in the world I would rather have shared it with.

Contents

Chapter 1

Introduction

Software is everywhere. In today's world, software flies spacecraft, monitors power plants, controls heart pacemakers, drives cars, and plays music. Unreliable software can make many aspects of life miserable and incur gigantic cost to the economy [57]. Therefore, companies and governments spend billions of dollars every year on ensuring the reliability of the software we use. This dissertation is part of the effort to reduce the cost of ensuring software reliability.

Testing is the primary approach to software reliability used in the software industry. Companies spend at least half of their development budget on testing [57]. Even when it is assisted by automated tools, software testing is often largely manual, which makes it expensive. To be effective, a suite of test inputs must exercise software in many different ways, and test engineers usually construct such test suites by hand. Often, software errors get revealed only for unusual inputs that rarely occur during normal use, and test engineers need significant skill in creating such inputs. This dissertation aims at lowering the cost of testing by automating the creation of good test inputs, thus reducing the time and effort required to discover errors.

White-box, or implementation-based, testing uses the program's code to create test inputs for the program. To gather information about the program under test, white-box testing techniques can analyze the program either statically (i.e., examine the text of the program), or dynamically (i.e., run the program and observe the execution). This dissertation presents novel dynamic white-box testing techniques and demonstrates their effectiveness.

Concolic testing (also known as *directed testing*, or *whitebox fuzzing*) is a new paradigm of dynamic white-box testing [17, 46, 102] that has been implemented in a number of tools [2, 16, 18, 48, 60]. This dissertation presents novel concolic testing techniques and tools. Concolic testing combines concolic execution (also known as *dynamic symbolic execution*) with execution-path enumeration.

Concolic execution blends concrete and symbolic execution [65]. Given a program and an input, concolic execution records how the input affects the control flow in the program. The result of concolic execution is a path constraint: a logic formula that is satisfied by the input and also by any other input that will drive the program's execution along the same execution path. Symbolic variables in the path constraint refer to bytes in the program's input.

Concolic testing enumerates execution paths in the program by specifying alternative paths as logic formulas and using a constraint solver to find inputs that correspond to those alternative paths. A constraint solver is a program that computes solutions to logic formulas in a given logic. Concolic testing tools often use solvers for linear arithmetic [33] or bit-vector arithmetic [42]. This dissertation presents concolic testing techniques that work by enhancing the expressiveness of the constraint solver.

The **key idea** of this dissertation is that the effectiveness and applicability of concolic testing can be greatly improved by using an expressive and efficient *string-constraint solver*, i.e., a solver for constraints on string variables. We present the design, implementation, and experimental evaluation of a novel string-constraint solver. Furthermore, we show novel techniques for two important problems in concolic testing: getting past input validation in programs with highly-structured inputs, and creating inputs that demonstrate security vulnerabilities in Web applications.

1.1 Hampi: A Solver for String Constraints

We present HAMPI [63], a novel solver for constraints over fixed-size string variables. HAMPI is designed to be used as a component in automatic software testing, analysis, and verification applications. Many such applications, including concolic testing, can be reduced to constraint generation followed by constraint solving. This separation of concerns 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 effective and sufficiently expressive off-the-shelf solvers for string constraints generated by analysis techniques for string-manipulating programs.

HAMPI constraints express membership in regular languages and fixed-size context-free languages. The constraints may contain context-free-language definitions, regular-language definitions and operations, and the membership predicate. Given a set of constraints, the solver outputs a string that satisfies all the constraints, or reports that the constraints are unsatisfiable.

1.2 Grammar-Based Concolic Testing

We present *grammar-based concolic testing* [45], a novel concolic testing technique for programs with structured inputs. Even though concolic testing exercises different execution paths systematically, the execution-path exploration is unguided. Often, some parts of the code are tested repeatedly, while others remain untested. In such cases, concolic testing spends its time exercising the easy-to-reach parts of the code, but it has a hard time generating inputs that exercise long execution paths in the program [45,63,72]. This is particularly a problem for programs that have highly-structured inputs — only a tiny fraction of all inputs exercise long paths and reach the core functionality, while a vast majority of inputs get quickly rejected. For example, a Web-service checks that a request is in a required format. Similarly, a compiler checks that the input satisfies a set of grammatical rules. However, the constraint solvers that underlie current concolic testing tools do not know the specification of the input format. Our insight is that, by using a solver enhanced with the input-format specification, concolic testing can explore non-error paths in the program's early stages.

Grammar-based concolic testing enhances the constraint solver with a specification of valid inputs, expressed as a formal grammar. This enhancement combines implementation-based testing with specification-based testing: the specification of the input format guides the concolic-testing tool towards harder-to-reach, later areas of the code. In grammar-based concolic testing, concolic execution generates grammar-based constraints and checks their satisfiability using a custom string-constraint solver. Our custom solver, when searching for a satisfying assignment, only considers assignments that satisfy the input format specification. Technically, the solver computes the intersection of the language of inputs that satisfy the constraint *and* the language of inputs that satisfy the specification.

1.3 Concolic Security Testing

We present *concolic security testing* [64], a novel concolic testing technique for finding security vulnerabilities in Web applications. SQL Injection (SQLI) and cross-site scripting (XSS) are widespread security vulnerabilities. To exploit them, the attacker crafts the input to the application to access or modify user data and execute malicious code. In the most serious attacks (called second-order XSS), an attacker can corrupt a database and cause subsequent users of the Web application to execute malicious code.

Concolic security testing systematically generates program inputs, runs each input using concolic execution (including dynamically tracing symbolic values through database accesses), and uses a string-constraint solver to find inputs that produce concrete exploits. Our technique is, as far as we know, the first to precisely addresses second-order XSS attacks.

Concolic security testing creates real attacks, has few false positives, incurs no runtime overhead for the deployed application, works without requiring modification of application code, and handles dynamic programming-language constructs.

1.4 Contributions and Results

This dissertation contributes novel techniques and tools that improve effectiveness and applicability of concolic testing.

1. HAMPI [63], a novel solver for string constraints (Chapter 3). HAMPI is expressive and efficient, and can be successfully applied to testing and analysis

of real programs. Our experiments used HAMPI in a static analysis for finding SQL injection vulnerabilities in Web applications, and compared HAMPI with another string-constraint solver. To facilitate further research in string-constraint solvers, we made HAMPI's source code, documentation, and the experimental data available.

2. Grammar-based concolic testing [45], a novel concolic testing technique for programs with structured inputs (Chapter 4). We have implemented the technique and evaluated it in two case studies.

   (i) Systematic testing of UNIX programs. Our experiments show that our technique 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.

   (ii) A large security-critical application, the JavaScript interpreter of Internet Explorer 7 (IE7). Our technique explores deeper program paths and avoids dead-ends due to non-parsable inputs. Compared to regular concolic testing, our technique increased coverage of the code generation module of the IE7 JavaScript interpreter from 53% to 81% while using three times fewer tests.

3. Concolic security testing [64], a novel concolic testing technique (Chapter 5). We created ARDILLA, a tool that implements this technique for PHP and applied ARDILLA to finding security vulnerabilities in Web applications. We evaluated the tool on five PHP applications and found 68 previously unknown vulnerabilities (23 SQLI, 33 first-order XSS, and 12 second-order XSS).

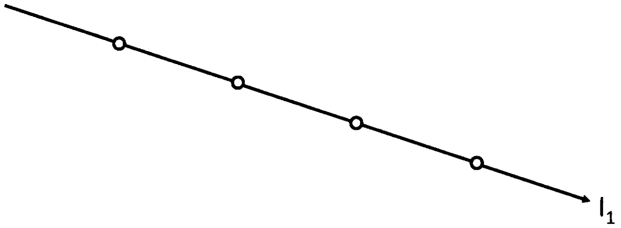
This dissertation begins with a description of concolic testing (Chapter 2). The next three chapters form the core of this dissertation and present the main contributions. Specifically, we present our solver for string constraints (Chapter 3), describe grammar-based concolic testing (Chapter 4), and describe concolic security testing (Chapter 5). We conclude with potential directions for future work (Chapter 6).

Chapter 2

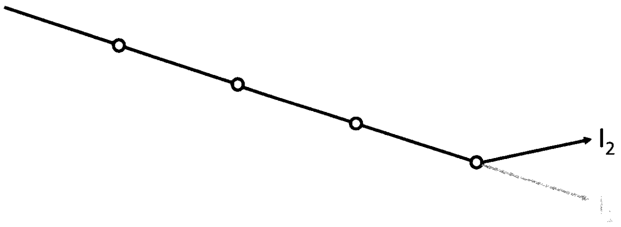
Concolic Testing

The purpose of concolic testing is to find program inputs that expose errors. The idea of concolic testing is to run the program on an input, observe the program execution on that input, and create additional inputs derived from the original. The additional inputs are perturbed versions of the original input and exercise execution-flow paths that are similar to that of the original input.

We illustrate the high-level idea of concolic testing. Running the program on the seed input (I_1) exercises a particular concrete execution path. In the following figure, the arrow denotes the execution path. The small circles denote the dynamic execution of conditional statements (e.g., `if`) that depend on the program input.

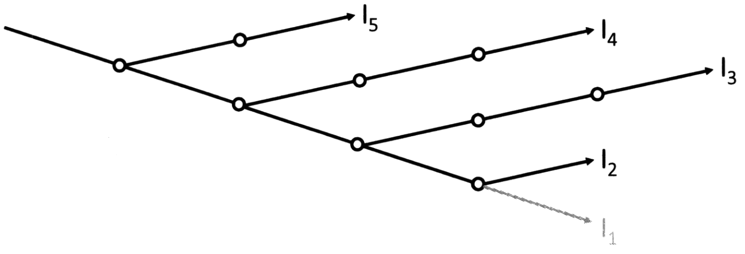


Concolic testing automatically generates inputs that exercise different execution-flow paths. Consider the following path that corresponds to a different input, I_2. This path agrees on the original path up to the last conditional, and then the execution takes *the opposite* branch. Concolic testing automatically finds such an input I_2.

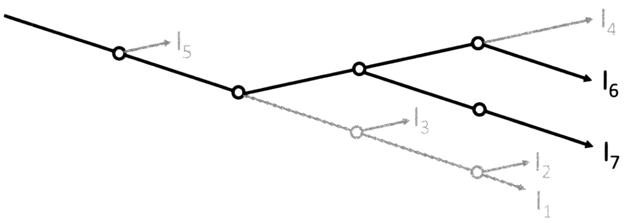


In the same way, from the original input I_1, concolic testing creates additional inputs I_3, I_4, and I_5 that exercise alternative execution-flow paths in the program.

Each path agrees on the original path up to the last conditional, and then the execution takes the opposite branch. Note that an alternative path need not be of the same length as the original path.



Each of the newly-created inputs is then executed, and thus concolic testing generates additional inputs. For example, executing I_4 allows concolic testing to generate I_6 and I_7 (for clarity, paths exercised by previously-generated inputs have been shortened). Thus, concolic testing runs the program multiple times, each time with a different input. Each newly-generated input may lead to the generation of additional inputs.



Concolic testing continues the process of running inputs and generating new inputs, until all execution-flow paths have been exercised, or until the time limit is reached.

In principle, concolic testing can exercise every feasible path in the program. In practice, however, concolic testing is usually incomplete because the number of feasible control paths may be astronomical (or even infinite) and because the precision of concolic execution, constraint generation, and solving is inherently limited. Nevertheless, concolic testing has been shown to be very effective in finding errors in programs [16, 17, 46, 102].

Algorithm

Now, we discuss the concolic testing algorithm in more detail. Concolic testing [17, 46, 102] combines concolic execution with execution-path enumeration. Given a sequential deterministic program $\mathcal{P}$ under test and a set of initial (seed) program *inputs*, concolic testing finds inputs for $\mathcal{P}$ that expose errors. The algorithm performs concolic execution of program $\mathcal{P}$ with each input, collects constraints generated during the execution, and generates new inputs by modifying and solving the constraints. Each such newly-generated input aims to exercise a different execution-flow path in $\mathcal{P}$. The algorithm repeats concolic execution and input generation, thus executing the program multiple times, each time with a different input. Each newly generated input may lead to the generation of additional inputs. The algorithm terminates when a testing time limit is reached or no more inputs can be generated (i.e., all execution paths in the code have been exercised).

```
   parameters: Program $\mathcal{P}$, seed inputs
   result    : Error reports for $\mathcal{P}$
 1 procedure concolicTesting($\mathcal{P}$, inputs):
 2 errors := $\emptyset$
 3 worklist := emptyQueue()
 4 enqueueAll(worklist, inputs)
 5 while not empty(worklist) and not timeExpired() do
 6     input := dequeue(worklist)
 7     errors := errors $\cup$ Run&Check($\mathcal{P}$, input)
 8     pc := concolicExecution($\mathcal{P}$, input)
 9     newInputs := createNewInputs(pc)
10     enqueueAll(worklist, newInputs)
11 return errors
```

Figure 2-1: Concolic testing algorithm.

First, the *worklist* of inputs is initialized with the seed *inputs* (line 4). Then, the algorithm loops until there are no more inputs to explore, or the time limit is reached (line 5). In the loop, the algorithm takes an input from the *worklist* (line 6), and checks whether running $\mathcal{P}$ with the input triggers a runtime error (line 7). The program is then executed concolically (procedure *concolicExecution*, explained below). The result of concolic execution is a *path constraint*, which concolic testing uses to generate new inputs (procedure *createNewInputs*, explained below).

Concolic Execution

Concolic execution (procedure *concolicExecution*) combines concrete and symbolic execution [65]. Concolic execution is also known as dynamic symbolic execution, because it is carried out while the program is running on a particular concrete input. Dynamic execution allows any imprecision in symbolic execution to be alleviated using concrete values: whenever symbolic execution does not know how to generate a symbolic expression for a runtime value, it can simplify the expression by using the concrete runtime value [46].

Concolic execution records how the input affects the control flow in the program. The result of concolic execution is a path constraint that is a logic formula

```
   parameters: Program $\mathcal{P}$, concrete input
   result    : Path condition pc
12 procedure concolicExecution($\mathcal{P}$, input):
13 path constraint pc := true
14 foreach instruction inst executed by $\mathcal{P}$ with input do
15     update the symbolic store
16     switch inst do
17         case input-dependent conditional statement
18             c := expression for the executed branch
19             $pc := pc \wedge c$
20         otherwise
21             if inst reads byte from input then
22                 mark input byte as symbolic variable
23 return pc
```

Figure 2-2: Concolic execution algorithm.

that is satisfied by the currently executed concrete input and any other concrete input that will drive the program's execution along the same control path. Symbolic variables in the path constraint refer to bytes in the program's input. The path constraint represents the equivalence class of all inputs that drive the execution of the program along the same control path as the original input. By memoizing the path constraints (or by using generation search [48]), concolic testing can guarantee executing at most one input from each equivalence class.

The algorithm keeps a *symbolic store* which is a map from concrete runtime values to symbolic expressions over symbolic variables. The algorithm updates the symbolic store whenever the program manipulates input data (line 15). At every conditional statement that involves symbolic expressions, the algorithm extends the current path constraint *pc* with an additional conjunct *c* that represents the branch of the conditional statement taken in the current execution (line 19). At every instruction that reads a byte from the input, a fresh symbolic variable is associated with the input byte (line 22).

Implementation strategies for concolic execution include source rewriting [18, 27,46,101,102], binary rewriting [84], and modifying the execution environment [2, 16,48,60,110]. Concolic execution tools exist for C [18,27,46,102], Java [60,101], .NET [110], and PHP [2,116].

Execution-Path Enumeration

Concolic testing creates new test inputs (procedure *createNewInputs* in Figure 2) by modifying the path constraint. For each prefix of the path constraint, the algorithm negates the last conjunct (line 28). A solution, if it exists, to such an alternative path constraint corresponds to an input that will execute the program along the prefix of the original execution path, but take the *opposite* branch of the condi-

```
   parameters: Path condition $pc$
   result     : New inputs
24 procedure $createNewInputs(pc)$:
25 $c_1 \land \ldots \land c_n := pc$
26 $inputs := \emptyset$
27 for $i := 1 \ldots n$ do
28     $pc := c_1 \land \ldots \land c_{i-1} \land \neg c_i$
29     $newInput := solve(pc)$
30     if $newInput \neq \bot$ then
31         $inputs := inputs \cup \{newInput\}$
32 return $inputs$
```

Figure 2-3: Creating new inputs in concolic testing.

tional statement corresponding to the last constraint in that prefix (assuming concolic execution and constraint solving have perfect precision, otherwise the actual execution may diverge from this path). The algorithm calls the constraint solver to find a concrete input that satisfies the alternative path constraint (line 29). If the constraint solver can find such a value (line 30), this new test input is generated (line 31).

Example

We illustrate concolic testing on an example program (from Godefroid et al. [48]). The program takes a 4-character array as input and has an error that can be exposed by only one input, `bad!` (for readability, we print the character array as a string). Finding this error without examining the implementation is hard — assuming 8-bit characters, only 1 input in 2^{8*4} (more than 4 billion) exposes the problem. Thus, randomly sampling the input space is unlikely to lead to quickly exposing the error. Concolic testing is implementation-based and finds the error-causing input very quickly, in a few program executions.

Concolic testing starts with an arbitrary seed input, and executes that seed input to collect constraints and create additional inputs. Then, concolic testing executes the program repeatedly with newly generated inputs. The process continues until all execution paths are explored, or until the time limit is reached.

Execution 1. Concolic testing starts with an arbitrary input, e.g., `good`, and performs concolic execution of the program on the input. Concolic execution marks each character in the input as symbolic. We denote the symbolic marks $i_0,\ldots,i_3$ (corresponding to `input[0]`,...,`input[3]`). When the execution reaches the first `if` statement (line 3), the path constraint (initially set to *true*) gets extended with conjunct $i_0 \neq$ `b`. This expression corresponds to the branch that is taken in the execution (in this case, the `else` branch of the conditional because the concrete value of i_0 is `g`). When the program execution finishes, the path constraint is:

```
1  void main(char input[4]) {
2   int count=0;
3   if (input[0] == 'b')
4     count++;
5   if (input[1] == 'a')
6     count++;
7   if (input[2] == 'd')
8     count++;
9   if (input[3] == '!')
10    count++;
11  if (count >= 3)
12    abort(); // error
13 }
```

$(i_0 \neq \texttt{b}) \wedge (i_1 \neq \texttt{a}) \wedge (i_2 \neq \texttt{d}) \wedge (i_3 \neq \texttt{!})$. Concolic testing generates four new inputs by systematically negating conjuncts in the path constraint and solving the resulting constraint.

pc$_1$ $(i_0 \neq \texttt{b}) \wedge (i_1 \neq \texttt{a}) \wedge (i_2 \neq \texttt{d}) \wedge (i_3 = \texttt{!})$ ⟶ goo!
pc$_2$ $(i_0 \neq \texttt{b}) \wedge (i_1 \neq \texttt{a}) \wedge (i_2 = \texttt{d})$ ⟶ godd
pc$_3$ $(i_0 \neq \texttt{b}) \wedge (i_1 = \texttt{a})$ ⟶ gaod
pc$_4$ $(i_0 = \texttt{b})$ ⟶ bood

For simplicity of exposition, we assume that all generated inputs are of the same length as the original input, and that the constraint solver prefers the concrete values from the original input to other concrete values. For example, there are many solutions to $(i_0 \neq \texttt{b}) \wedge (i_1 = \texttt{a})$, and we assume that the solver chooses gaod. This assumption does not change the technique. In fact, this assumption is commonly used in the implementation of concolic testing [16,48,60].

Each newly generated input explores a unique execution path. Each path shares a prefix with the original path, up to the conditional that corresponds to the negated conjunct, and then the new paths diverge. For example, the original input good executes the path (numbers indicate line numbers in the example program): 2,3,5,7,9,11. Input gaod executes the path 2,3,5,6,7,9,11. The two paths agree on the prefix 2,3,5 and then diverge.

Execution 2. Concolic testing executes one of the generated inputs, etc. bood. Concolic execution of this input results in path constraint: $(i_0 = \texttt{b}) \wedge (i_1 \neq \texttt{a}) \wedge (i_2 \neq \texttt{d}) \wedge (i_3 \neq \texttt{!})$. By systematically negating conjuncts in this path constraint and solving the resulting formulas, concolic testing creates three additional inputs.

pc$_4$ $(i_0 = \texttt{b}) \wedge (i_1 \neq \texttt{a}) \wedge (i_2 \neq \texttt{d}) \wedge (i_3 = \texttt{!})$ ⟶ boo!
pc$_5$ $(i_0 = \texttt{b}) \wedge (i_1 \neq \texttt{a}) \wedge (i_2 = \texttt{d})$ ⟶ bodd
pc$_6$ $(i_0 = \texttt{b}) \wedge (i_1 = \texttt{a})$ ⟶ baod

Execution $n \leq 16$. After a few more iterations, concolic testing generates the error-exposing input bad!. The exact number of iterations depends on the algorithm for selecting the next input to execute but, in this example, the number is not larger than 16, i.e., the number of feasible execution paths in the program. This is much fewer than 2^{64} possible inputs.

Chapter 3

Hampi: A Solver for String Constraints

Many automatic testing [18, 46, 102], analysis [50, 118], and verification [22, 23, 59] 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 [35, 86] and other theories [33, 42] continues to increase. Most of these solvers are aimed at propositional logic, linear arithmetic, or the theory of bit-vectors.

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 [12, 36,45], verifying correctness of program output [103], and finding security faults [41, 116] 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.

We designed and implemented HAMPI, a solver for constraints over fixed-size string variables. HAMPI constraints express membership in regular and fixed-size context-free languages[1]. HAMPI constraints may contain a fixed-size 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

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

to solve the intersection, containment, and equivalence problems for regular and fixed-size context-free languages.

A key feature of HAMPI is fixed-sizing of regular and context-free languages. Fixed-sizing makes HAMPI different from custom string-constraint solvers used in many testing and analysis tools [36]. As we demonstrate, for many practical applications, fixed-sizing 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 fixed-sizing. Furthermore, fixed-sizing makes the satisfiability problem solved by HAMPI tractable. This difference is similar to that between model-checking and bounded model-checking [11], and it has been previously explored in automated decision procedures for integer arithmetic [24] and relational logic [58].

HAMPI's input language can encode queries that help identify SQL injection attacks, 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`" (where `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).

HAMPI works in four steps: First, normalize 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 fixed-size string variable, and R is a regular expression. Second, translate 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, hand over the bit-vector constraints to STP [42], a constraint solver for bit-vectors and arrays. Fourth, if STP reports that the constraints are unsatisfiable, then HAMPI reports the same. Otherwise, STP reports 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 evaluated HAMPI in four experiments. We used HAMPI in testing and analysis applications and evaluated HAMPI's expressiveness and efficiency: (i) We used HAMPI in a tool for identifying SQL injection vulnerabilities in PHP Web applications using static analysis [114]. (ii) We compared HAMPI's performance to CFGAnalyzer [4], a solver for analyzing context-free grammars. (iii) We used HAMPI in ARDILLA [64], a tool for creating SQL injection attacks using dynamic analysis of PHP Web applications. (iv) We used HAMPI in Klee [16], a concolic-testing tool.

Our 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.

Contributions and Results

- A *decision procedure* for constraints over fixed-size string variables, supporting regular language membership, context-free language membership, and typical string operations such as 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`.

- Downloadable (from HAMPI website) *experimental data* that can be used as benchmarks for developing and evaluating future string-constraint solvers.

- Evaluation of application of HAMPI to a static analysis. We used HAMPI in the static analysis tool [114], which we applied 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 fixed-sizing the string constraints is not a handicap.

- Comparison of HAMPI's performance to another solver. We ran HAMPI and CFGAnalyzer [4] on 100 grammar intersection problems. On those benchmarks, HAMPI is, on average, 6.8× faster than CFGAnalyzer. Furthermore, HAMPI's speedup increased with the problem size.

We present the evaluation of HAMPI in ARDILLA and in Klee in the respective chapters (Section 5.4 and Section 4.4).

We introduce HAMPI using an example (Section 3.1), then present HAMPI's input format and solving algorithm (Section 3.2), discuss speed optimizations (Section 3.2.6), and present the experimental evaluation (Section 3.3). We finish with related work (Section 3.4).

3.1 Example: SQL Injection

SQL injections are a prevalent class of Web-application vulnerabilities. This section illustrates how an automated tool [64,116] could use HAMPI to detect SQL injection vulnerabilities and to produce attack inputs.

Figure 3-1 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 3-1 retrieves and displays messages for the given topic.

```
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 3-1: Fragment of a PHP program that displays messages stored in a MySQL database. This program is vulnerable to an SQL injection attack. Section 3.1 discusses the vulnerability.

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 an automated 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 3-1?" The HAMPI solver answers such questions, and creates strings that can be used as exploits.

HAMPI constraints can formalize the above question (Figure 3-2). Automated vulnerability-scanning tools [64,116] can create HAMPI constraints via either static or dynamic program analysis (we demonstrate both static and dynamic techniques in our evaluation in Section 3.3.1 and in Chapter 5). Specifically, a tool could create the HAMPI input of Figure 3-2 from analyzing the code of Figure 3-1.

We now discuss various features of the HAMPI input language that Figure 3-2 illustrates. (Section 3.2.1 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 5–10) introduces a *context-free grammar*, for a fragment of the SQL grammar of `SELECT` statements.

```
1  //string variable representing '$my_topicid' from Figure 3-1
2  var v : 12;     // size is 12 characters
3
4  //simple SQL context-free grammar
5  cfg SqlSmall := "SELECT " (Letter)+ " FROM " (Letter)+ " WHERE " Cond;
6  cfg Cond     := Val "=" Val | Cond " OR " Cond";
7  cfg Val      := (Letter)+ | "'" (LetterOrDigit)* "'" | (Digit)+;
8  cfg LetterOrDigit := Letter | Digit;
9  cfg Letter := ['a'-'z'] ;
10 cfg Digit  := ['0'-'9'] ;
11
12 //the SQL query '$sqlstmt' from line 3 of Figure 3-1
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'";
```

Figure 3-2: The HAMPI input that finds an attack vector that exploits the SQL injection vulnerability from Figure 3-1.

- 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 3-1.

- 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 SqlSmallFixedSize. 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 3-2 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 is possible to create other (usually related) attack inputs, such as 999' OR '1'='1.

3.2 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 (Figure 3-3):

1. Normalize the input constraints to a *core form* (Section 3.2.2).

2. Encode the constraints in bit-vector logic (Section 3.2.3).

3. Invoke the STP bit-vector solver [42].

4. Decode the results obtained from STP (Section 3.2.3).

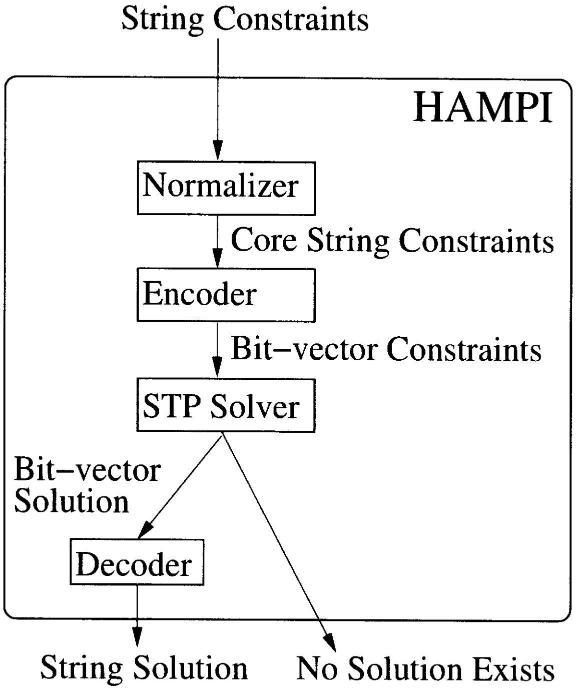


Figure 3-3: Schematic view of the HAMPI string solver. Section 3.2 describes the HAMPI solver.

3.2.1 Input Language for String Constraints

We discuss the salient features of HAMPI's input language (Figure 3-4) and illustrate them on examples. HAMPI's input language enables encoding of string constraints generated from typical testing and security applications. The language supports declaration of fixed-size string variables and constants, regular-language operations, membership predicate, and declaration of context-free and regular languages, temporaries and constraints.

Declaration of String Variable — `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 3-2 declares a variable `v` of size 12 characters.

Sometimes, an application of a string-constraint 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 3.2.6), or (ii) adjust the constraint to allow "padding" of the variable (e.g., using Kleene star to denote trailing spaces). It would be straightforward to extend HAMPI to permit specifying a size range, using syntax such as `var v:1..12`.

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
	|	**fixsize**(*Id* , *Int*)	CFG fixed-sizing
	|	**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 3-4: 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.

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 5–10 in Figure 3-2 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-sizing of a context-free language. Every regular language can be expressed using the first three of those operations [105].

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));
```

HAMPI allows construction of regular languages by fixed-sizing 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).

For example, in Figure 3-2, to describe the regular language that consists of all syntactically correct SQL strings of length 53 (according to the `SqlSmall` grammar) we would write `fixsize(SqlSmall, 53)`. Using the **fixsize** operator is more convenient than writing the regular expression explicitly.

HAMPI can infer the correct size and perform the fixed-sizing automatically. For example, in line 16 in Figure 3-2, HAMPI automatically infers that 53 is the correct size for `q` (sum of the size of the string-variable and the constant strings in the declaration of `q`: 12+40+1).

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 3-2 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 3-2 declares that the string value of the temporary variable `q` is in the regular language defined by `SqlSmallFixedSize`.

3.2.2 Core Form of String Constraints

After parsing and checking the input, HAMPI normalizes the string constraints to a core form (Figure 3-5). 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 HAMPI input 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).

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 3-5: The grammar of core string constraints. Nonterminals *Const* and *Var* have the usual definitions.

2. Expand all context-free grammar fixed-sizing expressions. The algorithm converts **fixsize** terms to regular expressions (see below for the algorithm). This step involves inference of sizes for cases in which the **fixsize** is not explicitly used. The inference is straightforward and we omit the details here.

3. Expand all regular-language declarations, i.e., replace each reference to a regular-language variable with the variable's definition.

Fixed-Sizing 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 [105].

3. Construct the regular expression that encodes all fixed-sized 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 subexpressions recursively and combine the subexpressions with a concatenation operator. Memoization of intermediate results (Section 3.2.6) makes this (worst-case exponential in k) process scalable.

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):

()[()() + (())] + [()() + (())]() + ([()() + (())])

3.2.3 Bit-vector Encoding and Solving

HAMPI encodes the core 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 (Figure 3-6). 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* $\in$ *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.

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 3-6: 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.

- 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.

3.2.4 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 (∨) 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 $\Sigma = \{\mathrm{T}, \mathrm{F}\}$ denote the alphabet. 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 $\mathrm{T}\Sigma\Sigma\cdots\Sigma$, is also of size n and has a T in the i^{th} slot for v_i (and F for $\neg v_i$), i.e.,

$$v_i \longrightarrow \underbrace{\overbrace{\Sigma\cdots\Sigma}^{i-1}\mathrm{T}\overbrace{\Sigma\cdots\Sigma}^{n-i}}_{n}$$

$$\neg v_i \longrightarrow \underbrace{\overbrace{\Sigma\cdots\Sigma}^{i-1}\mathrm{F}\overbrace{\Sigma\cdots\Sigma}^{n-i}}_{n}$$

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, in time polynomial in $k + r$, whether the witness belongs to each regular language.

3.2.5 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 ")";
val q := concat( "((" , v , "))" );
assert q in E;           // turns into constraint c1
assert q contains "())"; // turns into constraint c2
```

HAMPI follows the solving algorithm outlined in Section 3.2 (The alphabet of the regular expression or context-free grammar in a HAMPI input is implicitly restricted to the terminals specified):

step 1. Normalize constraints to core form, using the algorithm in Section 3.2.2. In this example, the size inference required to normalize the first `assert` gives size 6 for `q` (computed as 2+2+2 from the declarations of `v` and `q`). The core constraints are:

$$\begin{aligned}
\textbf{c1}:\quad \texttt{((}\,v\,\texttt{))} \;&\in\; \texttt{()}\big[\texttt{()()} \;+\; \texttt{(())}\big] + \\
&\quad\;\; \big[\texttt{()()} \;+\; \texttt{(())}\big]\texttt{()} \;+ \\
&\quad\;\; \texttt{(}\big[\texttt{()()} \;+\; \texttt{(())}\big]\texttt{)} \\
\textbf{c2}:\quad \texttt{((}\,v\,\texttt{))} \;&\in\; \big[\texttt{(} \;+\; \texttt{)}\big]\star \;\; \texttt{())} \;\; \big[\texttt{(} \;+\; \texttt{)}\big]\star
\end{aligned}$$

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

Formulas D_{1b} and D_{1c} for the remaining conjuncts are similar. The bit-vector formula that encodes **c1** is $C_1 = L_1 \wedge (D_{1a} \vee D_{1b} \vee D_{1c})$. 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. It reports the solution for v as "`)(`". (String "`()`" is another legal solution for v, but STP only finds one solution.)

3.2.6 Optimizations

Optimizations in Hampi aim at reducing computation time.

Memoization

Hampi stores and reuses partial results during the computation of fixed-sizing of context-free grammars (Section 3.2.2) and during the encoding of core constraints in bit-vector logic (Section 3.2.3).

Example. Consider the example from Section 3.2.5, i.e., fixed-sizing 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 3.2.5.

Constraint Templates

Constraint templates capture common encoded sub-expressions, modulo offset in the bit-vector. This optimization is related to the template mechanism proposed by Shlyakhter et al. [104], and to the sharing detection in the KodKod model finder [111]. During the bit-vector encoding step (Section 3.2.3), 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).

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 is $T(p) \equiv bv[p] = 41 \wedge bv[p + 1] = 40$, where p is the offset parameter. HAMPI then instantiates the template to $T(0)$ and $T(3)$.

As another example, consider **c1** in Section 3.2.5: 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.

Server Mode

The server mode improves HAMPI's efficiency on simple constraints and on repeated calls. 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.

3.3 Evaluation

We experimentally tested HAMPI's applicability to practical problems involving string constraints, and to compare HAMPI's performance and scalability to another string-constraint solver.

Experiments

1. We used HAMPI in a static-analysis tool [114] that identifies possible SQL injection vulnerabilities (Section 3.3.1).

2. We compared HAMPI's performance and scalability to CFGAnalyzer [4], a solver for bounded versions of context-free-language problems, e.g., intersection (Section 3.3.2).

Additionally, we used HAMPI in ARDILLA, a concolic testing tool that creates attacks on Web applications (Section 5.4), and in Klee, a concolic testing tool for UNIX programs (Section 4.4). We present these two experiments in the respective chapters.

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`.

3.3.1 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 [114] 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 [83]. 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 [114]. 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 [115]. 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.

HAMPI solves most queries quickly. Figure 3-7 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.

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 3-8 shows the solving time as a function of the grammar size, for all 1,367 inputs.

Our experimental results, obtained on a large dataset from a powerful static analysis and real Web applications, indicate that HAMPI's fixed-size solving algorithm is applicable to real problems.

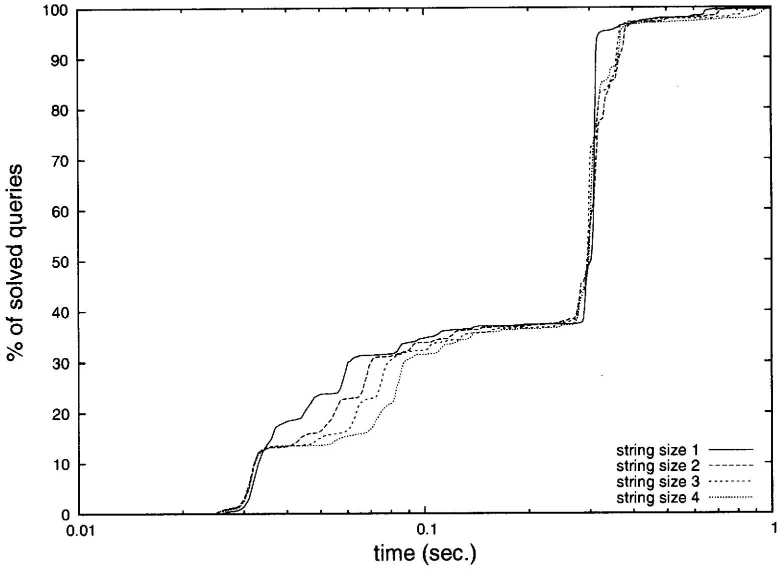


Figure 3-7: Percentage of queries solvable by HAMPI, in a given amount of time, on data from Wassermann and Su [114]. Each line represents a distribution for a different size of the string variable. All lines reach 99.7% at 1 second and 100% before 160 seconds. Section 3.3.1 describes the experiment.

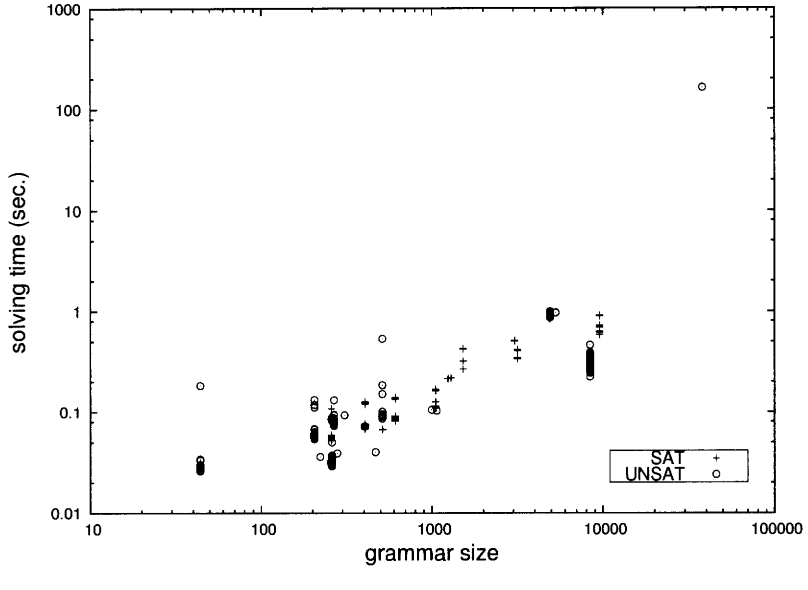


Figure 3-8: HAMPI solving time as function of grammar size (number of all elements in all productions), on 1,367 inputs from the Wassermann and Su dataset [114]. The size of the string variable was 4, the smallest at which HAMPI finds all satisfying assignments for the dataset. Each point represents an input; shapes indicate SAT/UNSAT. Section 3.3.1 describes the experiment.

3.3.2 Comparing Performance to CFGAnalyzer

We evaluated HAMPI's utility in analyzing context-free grammars, and compared HAMPI's performance to a specialized decision procedure, CFGAnalyzer [4]. 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[2] (released 3 December 2007) and configured the program according to the manual.

Experiment. 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 the experiments, both HAMPI and CFGAnalyzer searched for strings (of fixed length) from the intersection of 2 grammars. To avoid bias, we used CFGAnalyzer's own 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.

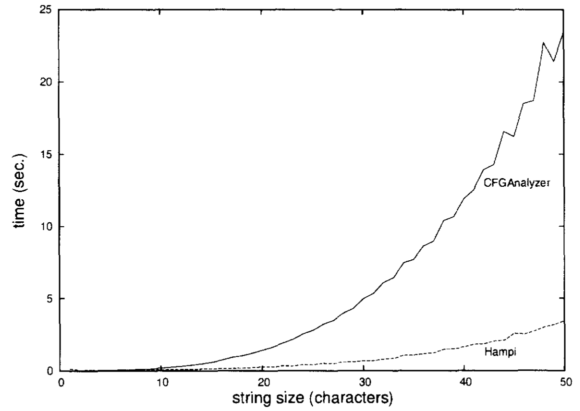


Figure 3-9: Solving time as a function of string size, on context-free-grammar intersection constraints. Results are averaged over 100 randomly-selected pairs of context-free grammars. Section 3.3.2 describes the experiment.

Results. HAMPI is faster than CFGAnalyzer for all sizes larger than 4 characters. Figure 3-9 shows the results averaged over all pairs of grammars. Importantly,

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

HAMPI's win 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.

HAMPI is faster also on grammar-membership constraints. We performed an additional experiment we: 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). The HAMPI website contains all experimental data and detailed results.

3.4 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 [35, 86], bit-vectors [42], quantified Boolean formulas [8, 10], and linear arithmetic [33]. 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 [66] 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 [38–40]. Fido [66, 67] is a higher-level formalism implemented on top of the MONA solver to concisely represent regular sets of strings and trees.

Word equations [12,96] describe equality between two strings that contain string variables. Rajasekar [96] proposes a logic programming approach that includes constraints on individual words. His solver handles concatenation but not regular language membership. Bjørner et al. [12] describe a constraint solver for word queries over a variety of operations, and translate string constraints to the language of the Z3 solver [33]. If there is a solution, Z3 returns a finite bound for the set of strings, that is then explored symbolically. Alonso et al. [1] describe a solver for word equations, based on genetic algorithms. However, unlike HAMPI, these tools do not support context-free grammars directly.

Hooimeijer and Weimer [54] 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 fixed-size context-free grammars.

Custom String Solvers

Many analyses use custom solvers for string constraints [7,15,20,36,41,45,83,114–116]. All of these approaches include some implementation for language intersection and language inclusion; most, similarly to 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. [20] 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. [41] also use an automata-based method to solve string constraints. Ruan et al. [97] 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 Wassermann et al. [114], the solver by Emmi et al. [36] is closest to HAMPI. 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 such as HAMPI, is warranted.

Theoretical Work on String Constraints

A variety of problems involve strings constraints, and there is an extensive literature on the theoretical study of these problems [31,32,49,73,91,93,95,100]. Our work is focused on efficient techniques for a practical string-constraint solver that is usable as a library and is sufficiently expressible to support a large variety of applications.

3.5 Conclusion

We presented HAMPI, a solver for constraints over fixed-size string variables. HAMPI constraints express membership in regular and fixed-size context-free languages. HAMPI constraints may contain a fixed-size 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 fixed-size 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-constraint solver such as HAMPI, builders of analysis and testing tools can save significant development effort, and improve the effectiveness of their tools.

Chapter 4

Grammar-Based Concolic Testing

Grammar-based concolic testing enhances concolic testing with a grammar-based specification of valid inputs. The effectiveness of concolic testing is limited when testing programs with highly-structured inputs. Such programs (e.g., compilers and interpreters) process their inputs in stages, such as lexing, parsing, and evaluation. Due to the enormous number of control paths in early processing stages, concolic testing rarely reaches parts of the program beyond these first stages. For example, there are many possible sequences of blank-spaces, tabs, carriage-returns, that separate tokens in most structured languages, and each sequence corresponds to a different control path in the lexer. In fact, for programs with highly-structured inputs, concolic testing is often not much better than simple *fuzzing*, i.e., creating test inputs by randomly modifying well-formed inputs.

Concolic execution itself may be defeated already in the first processing stages. For example, lexers often detect language keywords by comparing the pre-computed, hard-coded hash values of keywords with hash values of strings read from the input. This effectively prevents concolic testing from ever generating input strings that match those keywords since hash functions cannot easily be inverted (e.g., given a constraint `x == hash(y)` and a value for `x`, one cannot compute a value for `y` that satisfies this constraint).

Grammar-based concolic testing uses a grammar as an input-format specification. In grammar-based concolic testing concolic, execution generates grammar-based constraints, whose satisfiability is checked using a custom grammar-based constraint solver. The algorithm has two key components:

1. Generation of higher-level symbolic constraints, expressed in terms of symbolic grammar tokens returned by the lexer, instead of the traditional [18, 46, 48] symbolic bytes read as input.

2. A custom solver for constraints on symbolic grammar tokens. The solver looks for solutions that satisfy the constraints *and* are accepted by a given (context-free) grammar.

Our technique offers two key benefits: (i) The technique avoids dead-ends (error paths) in the lexer and parser; assuming the grammar accepts only parsable inputs, the technique generates only parsable inputs. (ii) The grammar-based constraint solver can *complete* a partial set of token constraints into a fully-defined valid input, hence avoiding exploring many possible non-parsable completions. By restricting the search space to valid inputs, grammar-based concolic testing can exercise longer paths, and focus the search on the harder-to-test, deeper processing stages.

Contributions and Results

- Grammar-based concolic testing, a novel algorithm for effective concolic testing of programs with highly-structured inputs.
- Evaluation of grammar-based concolic testing. We have implemented the technique and evaluated it in two case studies.
  (i) Concolic testing of UNIX programs. Our technique 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.
  (ii) A large security-critical application, the JavaScript interpreter of Internet Explorer 7 (IE7). Compared to regular concolic testing, our technique increased coverage of the code generation module of the IE7 JavaScript interpreter from 53% to 81% while using only one third of the test inputs.

We introduce grammar-based concolic testing using an example (Section 4.1), then explain the technique (Section 4.2). We present two experimental studies: JavaScript interpreter (Section 4.5), and concolic testing of UNIX programs (Section 4.4). We finish with related work (Section 4.6).

4.1 Example: Concolic Testing an Interpreter

We present grammar-based concolic testing on an example interpreter. Consider the JavaScript interpreter sketched in Figure 4-1 and the JavaScript grammar partially defined below. (*Nonterminals* are italicized, `terminals` are in tele-type font. Symbol ϵ denotes the empty string. The starting nonterminal is *FunDecl*.)

```
FunDecl   ::=  function id ( Formals ) FunBody
FunBody   ::=  { SrcElem* }
Formals   ::=  id (, Formals)*
SrcElem   ::=  ...
...
```

By tracking the tokens returned by the lexer (i.e., the function `nextToken`, line 3 in Figure 4-1) and considering those as symbolic inputs, our concolic-testing

```
// Lexer: Reads and returns next token from file.
// Terminates on erroneous inputs.
Token nextToken(){
  ...
  readInputByte();
  ...
}

// Parser: Parses the input file, returns parse tree.
// Terminates on erroneous inputs.
ParseTree parse(){
  ...
  Token t = nextToken();
  ...
}

void main(){
  ...
  ParseTree t = parse();
  ...
  Bytecode code = codeGen(t);
  ... // Execute code
}
```

Figure 4-1: Sketch of an interpreter. The interpreter processes the inputs in stages: lexer (function `nextToken`), parser (function `parse`), and code generator (function `codeGen`). Next, the interpreter executes the generated bytecode (omitted here).

algorithm generates constraints in terms of such tokens. For example, running the interpreter on the valid input

```
function f(){ }
```

may correspond to the sequence of symbolic token constraints

$\text{token}_0 = \texttt{function}$
$\text{token}_1 = \texttt{id}$
$\text{token}_2 = \texttt{(}$
$\text{token}_3 = \texttt{)}$
$\text{token}_4 = \{$
$\text{token}_5 = \}$

Negating the fourth constraint in this path constraint leads to the new sequence of constraints:

$\text{token}_0 = \texttt{function}$
$\text{token}_1 = \texttt{id}$
$\text{token}_2 = \texttt{(}$
$\text{token}_3 \neq \texttt{)}$

There are many ways to satisfy this constraint, but most solutions lead to non-parsable inputs. In contrast, our grammar-based constraint solver can directly conclude that the only way to satisfy this constraint *while generating a valid input* according to the grammar is to set

token_3 = `id`

and to complete the remainder of the input with, for example,

token_4 =)
token_5 = {
token_6 = }

Thus, the generated input that corresponds to this solution is

```
function f(id){ }
```

where `id` can be any identifier.

Similarly, the grammar-based constraint solver can immediately prove that negating the third constraint in the previous path constraint, thus leading to the new path constraint

token_0 = `function`
token_1 = `id`
token_2 ≠ `(`

is *unsolvable*, i.e., there are no inputs that satisfy this constraint and are recognized by the grammar. Grammar-based concolic testing prunes in one iteration the entire sub-tree of lexer executions corresponding to all possible non-parsable inputs matching this case.

4.2 Concolic Testing of Programs with Structured Inputs

In this section, we introduce grammar-based concolic testing. We then discuss how to check grammar-based constraints for context-free grammars (Section 4.2.1). Finally, we discuss additional aspects of our approach and some of its limitations (Section 4.2.2).

Grammar-based concolic testing (Figure 4-2) extends concolic testing (Section 2) as follows:

- The new algorithm requires a grammar $\mathcal{G}$ that describes valid program inputs (line 1).
- Grammar-based concolic testing marks tokens returned from a tokenization function such as `nextToken` in Figure 4-1 as symbolic (line 18). Thus, grammar-based concolic testing associates a symbolic variable with each token[1], and symbolic execution tracks the influence of the tokens on the control path taken by the program $\mathcal{P}$.

[1] Symbolic variables could also be associated with other values returned by the tokenization function for specific types of tokens, such as the string value associated with each identifier, the numerical value associated with each number, etc.

```
parameters: Program $\mathcal{P}$, seed inputs, $\underline{\text{grammar } \mathcal{G}}$
result    : Error reports for $\mathcal{P}$
Procedure grammarBasedConcolic($\mathcal{P}$, inputs$\underline{, \mathcal{G}}$):
errors := $\emptyset$
worklist := emptyQueue()
enqueueAll(worklist, inputs)
while not empty(worklist) and not timeExpired() do
    input := dequeue(worklist)
    errors := errors $\cup$ Run&Check($\mathcal{P}$, input)
    pc := concolicExecution($\mathcal{P}$, input)
    newInputs := createNewInputs(pc$\underline{, \mathcal{G}}$)
    enqueueAll(worklist, newInputs)
return errors

Procedure symbolicExecution($\mathcal{P}$, input):
path constraint pc := true
foreach instruction inst executed by $\mathcal{P}$ with input do
    update the symbolic store
    switch inst do
        case $\underline{\text{return from tokenization function}}$
            $\underline{\text{mark token as symbolic variable}}$
        case input-dependent conditional statement
            $c$ := expression for the executed branch
            $pc := pc \wedge c$
        otherwise
            if $\underline{false}$ $\wedge$ inst reads byte from input then
                mark input byte as symbolic variable;
return pc

Procedure createNewInputs(pc$\underline{, \mathcal{G}}$):
$c_1 \wedge \ldots \wedge c_n := pc$
inputs := $\emptyset$
for $i := 1 \ldots n$ do
    $pc := c_1 \wedge \ldots \wedge c_{i-1} \wedge \neg c_i$
    newInput := solve(pc$\underline{, \mathcal{G})}$
    if newInput $\neq \perp$ then
        inputs := inputs $\cup$ {newInput}
return inputs
```

Figure 4-2: Grammar-based concolic testing. Changes from concolic testing are underlined. Grammar-based concolic testing requires the constraint solver (auxiliary procedure *solve*) to handle grammar constraints (Figure 4-3).

- The algorithm uses the grammar $\mathcal{G}$ to require that the new input satisfy the alternative path constraint and be in the language accepted by the grammar (line 31). As the examples in the introduction illustrate, this requirement gives two advantages to grammar-based concolic testing: it enables pruning of the search tree corresponding to invalid inputs (i.e., inputs that are not accepted by the grammar), and it enables direct completion of satisfiable token constraints into valid inputs.

4.2.1 Context-free Constraint Solver

This section presents an algorithm for the constraint solver that is invoked in line 31 of Figure 4-2. The constraint solver implements the procedure *solve* and computes a language intersection: the solver checks whether the language $L(pc)$ of inputs satisfying the path constraint pc contains an input that is in the language accepted by the grammar $\mathcal{G}$. By construction, the language $L(pc)$ is regular, as we discuss later in this section. If $\mathcal{G}$ is context-free, then language intersection with $L(pc)$ can be computed. If $\mathcal{G}$ is context-sensitive, then a sound and complete procedure for computing language intersection may not exist (but approximations are possible). In what follows, we assume that $\mathcal{G}$ is context-free.

A *context-free constraint solver* takes as inputs a context-free grammar $\mathcal{G}$ and a regular expression R, and returns either a string $s \in L(\mathcal{G}) \cap L(R)$, or $\perp$ if the intersection is empty.

The HAMPI string-constraint solver (Chapter 3) can be used as a context-free constraint solver. However, the solver described in this section uses a simpler algorithm that works well in practice. Moreover, HAMPI is limited to fixed-size solutions, while the algorithm described here can complete partial solutions of any length. For our case studies, we used both HAMPI (Section 4.4) and the algorithm described here (Section 4.5).

Figure 4-3 presents an algorithm for a context-free constraint solver. The algorithm iteratively "unrolls" the productions for the start symbol (i.e., exposes prefixes of terminal symbols for each production), and "prunes" productions that do not conform to the regular expression R. Thus, the algorithm exploits the fact that, by construction, any regular language R constrains only the first n tokens returned by the tokenization function, where n is the highest index i of a token variable token$_i$ appearing in the constraint represented by R.

The algorithm starts by converting the path constraint pc into a regular expression R (line 2). The language $L(pc)$ is regular. We assume that the set of tokens that can be returned by the tokenization function is finite. Therefore, all token variables $token_i$ have a finite range, and satisfiability of any constraint on a finite set of token variables is decidable. Given any such constraint pc, one can sort its set of token variables $token_i$ by their index i, representing the total order by which they have been created by the tokenization function, and build a regular expression (language) R representing $L(pc)$ for that constraint pc.

Line 3 eliminates recursion for the start symbol S. This is done by duplicating productions for the start symbol and renaming the start symbol in the duplicated

```
   parameters: Path constraint $pc$, grammar $\mathcal{G}$ with start symbol $S$
   result    : String $s \in L(pc) \cap L(\mathcal{G})$ or $\perp$
1  Procedure solve($pc$, $\mathcal{G}$):
2  $R$ := buildConstraint($pc$)
3  $\mathcal{G}'$ := eliminate recursion on starting nonterminal $S$
4  $n$ := highest index $i$ of $token_i$ variable in $R$
5  for $i := 1 \ldots n$ do
       // loop invariant:  first $i-1$ symbols in productions
          for $S$ are terminals
6      let $c_i$ denote the constraint in $R$ on variable $token_i$
7      worklist $W$ := productions for $S$ in $\mathcal{G}'$
8      while not empty($W$) do
9          $prod$ := dequeue($W$)
10         $S_i$ := $i^{th}$ symbol in $prod.rhs$
11         if $S_i$ does not exist then
12             prune: remove $prod$ from $\mathcal{G}'$
13         else if $S_i$ is nonterminal $N$ then
14             unroll: remove $prod$ from $\mathcal{G}'$, add copies of $prod$ to $W$ and $\mathcal{G}'$, with
               $S_i$ expanded using all productions for $N$ in $\mathcal{G}'$
15         else
16             prune: remove $prod$ from $\mathcal{G}'$ if $S_i$ does not satisfy $c_i$
17 if $L(\mathcal{G}') = \emptyset$ then
18     return $\perp$
19 else
20     return generate $s$ from $\mathcal{G}'$
```

Figure 4-3: Procedure *solve(pc, $\mathcal{G}$)* implements a context-free constraint solver. The auxiliary function *buildConstraint(pc)* converts the path constraint *pc* to a regular expression. Notation *prod.rhs* denotes the right hand side of the production *prod*.

productions and in the right-hand sides of the original productions for S. The algorithm employs a simple unroll-and-prune approach: in the i^{th} iteration of the main loop (line 5), the algorithm unrolls the right-hand sides of productions to expose a $0 \ldots i$ prefix of terminals (line 14), and prunes those productions that violate the constraint c_i on the i^{th} token variable token$_i$ in the regular expression R (line 16). During each round of unrolling and pruning, the algorithm uses the worklist W to store productions that have not yet been unrolled and examined for conformance with the regular expression.

Finally, the algorithm produces a result string. After the unrolling and pruning, the algorithm checks emptiness [55] of the resulting language $L(\mathcal{G}')$ and generates a string s from the intersection grammar $\mathcal{G}'$ (line 20). For speed, our implementation uses a bottom-up strategy that generates a string with the shortest derivation tree (i.e., tree of smallest height) for each nonterminal in the grammar, by combining the strings from the right-hand sides of productions for nonterminals. This strategy is fast due to memoizing strings during generation. Section 4.2.2 discusses alternatives and limitations of the algorithm in Figure 4-3.

Solving Example

We illustrate the algorithm on an example, a simplified S-expression grammar. Starting with the initial grammar, the algorithm unrolls and prunes productions given a regular path constraint. The grammar is (S is the start symbol, *nonterminals* are italicized)

$$\begin{array}{lcl} S & ::= & \texttt{(let ((id } S\texttt{))}\ S\texttt{)} \mid \texttt{(}Op\ S\ S\texttt{)} \mid \texttt{num} \mid \texttt{id} \\ Op & ::= & + \mid - \end{array}$$

and the regular path constraint R (created by *buildConstraint(pc)*, line 2) is

$$\begin{array}{lcl} token_1 & \in & \{(\} \\ token_2 & \in & \{+\} \\ token_3 & \in & \{(\} \\ token_4 & \in & \{(,), \texttt{num}, \texttt{id}, \texttt{let}\} \end{array}$$

Before the main iteration (line 5), the grammar is:

$$\begin{array}{lcl} S' & ::= & \texttt{(let ((id } S'\texttt{))}\ S'\texttt{)} \mid \texttt{(}Op\ S'\ S'\texttt{)} \mid \texttt{num} \mid \texttt{id} \\ Op & ::= & + \mid - \\ S & ::= & \texttt{(let ((id } S'\texttt{))}\ S'\texttt{)} \mid \texttt{(}Op\ S'\ S'\texttt{)} \mid \texttt{num} \mid \texttt{id} \end{array}$$

Next, the main iteration begins. The first conjunct in the grammar constraint is $token_1 \in \{(\}$, therefore the algorithm (line 16) removes the last two productions for the start symbol S. The result is the following grammar (execution is now back at the top of the loop in line 5).

$$\begin{array}{lcl} S' & ::= & \texttt{(let ((id } S'\texttt{))}\ S'\texttt{)} \mid \texttt{(}Op\ S'\ S'\texttt{)} \mid \texttt{num} \mid \texttt{id} \\ Op & ::= & + \mid - \\ S & ::= & \texttt{(let ((id } S'\texttt{))}\ S'\texttt{)} \mid \texttt{(}Op\ S'\ S'\texttt{)} \end{array}$$

In the next iteration of the **for** loop, the algorithm examines the second conjunct in the regular path constraint, $token_2 \in \{+\}$. The algorithm prunes the first production rule from S since `let` does not match + (line 16), and then expands the nonterminal Op in the production $S ::= (Op\ S'\ S')$ (line 14). The production is replaced by two productions, $S ::= (+\ S'\ S')$ and $S ::= (-\ S'\ S')$, which are added to the worklist W. The grammar G' is then

$$\begin{array}{lcl} S' & ::= & \texttt{(let ((id } S'\texttt{))}\ S'\texttt{)} \mid \texttt{(}Op\ S'\ S'\texttt{)} \mid \texttt{num} \mid \texttt{id} \\ Op & ::= & + \mid - \\ S & ::= & (+\ S'\ S') \mid (-\ S'\ S') \end{array}$$

In the next iteration of the **while** loop, the second of the new productions is removed from the grammar (line 16) because it violates the grammar constraint. After the removal, the execution is now again at the top of the loop in line 5.

$$\begin{array}{lcl} S' & ::= & (\texttt{let}\ ((\texttt{id}\ S'))\ S') \mid (Op\ S'\ S') \mid \texttt{num} \mid \texttt{id} \\ Op & ::= & + \mid - \\ S & ::= & (+\ S'\ S') \end{array}$$

After 2 more iterations of the **for** loop, the algorithm arrives at the final grammar

$$\begin{array}{lcl} S' & ::= & (\texttt{let}\ ((\texttt{id}\ S'))\ S') \mid (Op\ S'\ S') \mid \texttt{num} \mid \texttt{id} \\ Op & ::= & + \mid - \\ S & ::= & (+\ (\texttt{let}\ ((\texttt{id}\ S'))\ S')\ S') \end{array}$$

As the last two steps, the algorithm checks that $L(\mathcal{G}') \neq \emptyset$ (line 17) and generates a string s from the final grammar $\mathcal{G}'$ for the intersection of $\mathcal{G}$ and R (line 20). Our bottom-up strategy generates the string $(+\ (\texttt{let}\ ((\ \texttt{id}\ \texttt{num}\ ))\ \texttt{num})\ \texttt{num})$. From this string of tokens, our tool generates a matching string of input bytes by applying an application-specific de-tokenization function.

4.2.2 Discussion and Limitations

Computing Language Intersection

Computing the intersection of a context-free grammar with a regular expression is a well-known problem. A standard polynomial-time algorithm consists in translating the grammar into a pushdown automaton, translating the regular expression into a finite-state automaton, computing the product of these two automata to obtain another pushdown automaton, and finally translating the resulting pushdown automaton back into a context-free grammar. Alternatively, the intersection can be computed without the explicit automata conversion [114], by an adaptation of the context-free reachability algorithm [80]. The HAMPI solver (Chapter 3) can also be used.

The unroll-and-prune algorithm from Section 4.2.1 is simpler than HAMPI or the context-free reachability. Moreover, unlike HAMPI, the algorithm can be used for completing a partial solution of any length (HAMPI only finds fixed-size solutions). The algorithm (Section 4.2.1) exploits the structure of the regular language that describes the path constraint on only the first n tokens returned by the tokenization function, where n is the highest index i of a token variable token$_i$ appearing in the constraint represented by R. This algorithm is not polynomial in general, but performs satisfactorily in practice for n around 50–60. By exploiting the special structure of the regular language, this algorithm can be faster than HAMPI (but it is less general). If the grammar is left-recursive, the algorithm in Figure 4-3 may not terminate. However, context-free grammars for file formats and programming languages are rarely left-recursive, and left-recursion can be efficiently removed [85].

Approximate Grammars

Grammar-based concolic testing can be used with approximate grammars. Let us call an input *parsable* if the parser successfully terminates when run on that input. If the grammar accepts all parsable inputs or over-approximates the set of parsable inputs, then the algorithm in Figure 4-2 is sound: it does not prune any of the feasible paths for which the parser successfully terminates.

In practice, grammar membership only approximates validity. The set of valid inputs specified by a grammar is an approximation of the set of parsable inputs. Indeed, parsers typically implement additional validation (e.g., simple type-checking) that is not part of the grammar description of the language. Other grammars may have some "context-sensitive behaviors" (as in protocol description languages, in which a variable size parameter k is followed by k records), that are omitted or approximated in a context-free or regular manner. Other grammars, especially for network protocols, are simplified representations of valid inputs, and do not require the full power of context-sensitivity [13,29,90].

Domain Knowledge

Grammar-based concolic testing requires a limited amount of domain knowledge. Specifically, applying the technique requires identifying the grammar of the input format and the tokenization function. Also a de-tokenization function needs to be provided, that generates byte strings from token strings generated by the context-free constraint solver.

We believe these are not severe practical limitations. Indeed, grammars are typically available for many input formats, and identifying the tokenization function is, in our experience, rather easy, even in unknown code, provided that the source code is available or that the tokenization function has a standard name, such as `token`, `nextToken`, `scan`, etc. For example, we found the tokenization function in the JavaScript interpreter of Internet Explorer 7 in a matter of minutes, by looking for commonly used names in the symbol table.

Lexer and Parser Errors

Using a grammar to filter out invalid inputs may reduce code coverage in the lexer and parser themselves, since the grammar explicitly prevents the execution of code paths handling invalid inputs in those stages. For testing those stages, traditional concolic testing can be used. Moreover, our experiments (Sections 4.5–4.4) indicate that grammar-based concolic testing does not decrease coverage in the lexer or parser.

Grammar-based concolic testing approach uses the actual lexer and parser code of the program under test. Indeed, removing these layers and using automatically generated software stubs simulating those parts may feed unrealistic inputs to the rest of the program.

4.3 Grammar-Based Concolic Testing Case Studies

We performed two case studies to evaluate grammar-based concolic testing. For the case studies, we implemented our technique in two existing concolic-testing tools, Klee [16] (Section 4.4) and SAGE [48] (Section 4.5). For each case study, we designed experiments with the following goals:

- Measure whether test inputs generated by grammar-based concolic testing are effective in exercising long execution paths in the program, i.e., reaching beyond the lexer and parser.
- Compare grammar-based concolic testing to other approaches, in particular concolic testing.

The results of our case studies show that grammar-based concolic testing is a general and powerful technique. In each case study, we were able to successfully implement grammar-based concolic testing in a tool with which we were initially unfamiliar (though we did interact with the tools' authors). Moreover, in each case study, grammar-based concolic testing significantly improved the effectiveness of the testing tool (in terms of code coverage), for programs with highly-structured inputs.

4.4 Case Study: Concolic Testing of UNIX Programs

We enhanced Klee [16], a concolic testing tool for UNIX programs, with grammar-based concolic testing. We aimed to improve Klee's ability to create valid test cases for programs that accept highly structured inputs.

In this study, we used the HAMPI string-constraint solver (Chapter 3) and not the solver from Section 4.2.1. HAMPI was more suited for this study because: (i) Klee, like HAMPI, works on fixed-size string input, and (ii) Klee is integrated with the STP [42] solver and HAMPI encodes string constraints in STP.

The rest of this section describes our experiments and discusses the results.

Summary of Results

Compared to concolic testing, grammar-based concolic testing 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 testing tools.

4.4.1 Experiment

We compared the coverage achieved and numbers of legal (and rejected) inputs generated by running Klee with and without HAMPI string constraints. While previous studies used custom-made string-constraint solvers, we are able to apply HAMPI as an "off-the-shelf" solver for Klee without modifying Klee at all.

Klee provides an Application Programming Interface (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 [72]). `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:

1. Automatically convert its Yacc specification into HAMPI's input format (described in Section 3.2.1), 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 [48]) 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 3.2.3).

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.

Program	ELOC	input size	strategy	line coverage %		generated inputs	
		(bytes)		total	parser	legal/all	(%)
`cueconvert`	939	28	**concolic**	32	21	0/14	(0%)
			concolic+grammar	51	77	146/146	(100%)
			combined	56	79	146/160	(91%)
`logictree`	1,492	7	**concolic**	31	12	70/110	(64%)
			concolic+grammar	63	65	98/98	(100%)
			combined	67	65	188/208	(81%)
`bc`	1,669	6	**concolic**	27	12	2/27	(5%)
			concolic+grammar	43	40	198/198	(100%)
			combined	47	43	200/225	(89%)

Table 4.1: 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 the 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. Klee generates a new input only when it covers new lines that previous inputs have not yet covered (to create minimal test suites); the total number of explored paths is usually hundreds of times greater than the number of generated inputs. Rows **concolic** show results for runs of Klee without a HAMPI grammar. Rows **concolic+grammar** show results for runs of Klee with a HAMPI grammar. Rows **combined** show accumulated results for both kinds of runs. Section 4.4 describes the experiment.

We made four 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,
- the numbers of inputs (test cases) generated, and
- the number of *legal* inputs (i.e., not rejected by the program as a parse error).

4.4.2 Results

Table 4.1 summarizes our experimental setup and results. 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.

Grammar-based concolic testing 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.

Grammar-based concolic testing can be combined with traditional concolic testing. Using *both* sets of inputs (**combined** rows) 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 (**concolic** rows) covered lines dealing with processing parse errors, whereas the inputs generated with the grammar (**concolic+grammar** rows) never had parse errors and covered core program logic. Thus, combining test suites is useful for testing both error and regular execution paths.

Grammar-based concolic testing uncovered unknown 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).

4.5 Case Study: JavaScript Interpreter

We extended SAGE [48], a concolic-testing tool for Windows programs, with grammar-based concolic testing. We aimed to improve SAGE's ability to create valid test cases for programs that accept highly structured inputs. We focused the case study on the JavaScript interpreter in Internet Explorer 7. In addition to the goals stated in Section 4.3, our experiments had the following goals:

- Measure how the set of inputs generated by each technique compares. In particular, do inputs generated by grammar-based concolic testing exercise the program in ways that other techniques do not?

- Measure the effectiveness of token-level constraints in preventing path explosion in the lexer.

- Measure the performance of the grammar-based constraint solver of Section 4.2.1 with respect to the size of test inputs.

- Measure the effectiveness of the grammar-based approach in pruning the search tree.

The rest of this section describes our experiments and discusses the results. Naturally, because they come from a limited sample, these experimental results need to be taken with caution. However, our evaluation is extensive and performed with a large, widely-used JavaScript interpreter, a representative "real-world" program.

Summary of Results. Compared to regular concolic testing, our technique increased coverage of the deepest of analyzed modules from 53% to 81% while using three times fewer tests. At least 27% of instructions covered by grammar-based concolic testing were not covered by any other automated strategy. Using the grammar-based constraint solver was effective: in 29% of cases, using the solver enabled pruning the search tree. The solver was efficient: at most 58% of total time was spent in the solver (comparable to concolic testing's 62%).

4.5.1 Subject Program

We performed the experiments with the JavaScript interpreter embedded in the Internet Explorer 7 Web-browser. We ran the interpreter with no source modifications. The total size of the JavaScript interpreter is 113562 machine instructions. The interpreter has three main modules: code generator, parser, and lexer. The code generator (3693 instructions) is the "deepest" of the examined modules, i.e., every input that reaches the code generator also reaches the other two modules (but the converse does not hold). The parser (18535 instructions) and lexer (10410 instructions) are equally deep, because the parser always calls the lexer.

We used the official JavaScript grammar[2] with 189 productions, 82 terminals (tokens), and 102 nonterminals.

4.5.2 Test-Generation Strategies

We evaluated the following test input generation strategies, to compare them to grammar-based concolic testing.

[2] `http://www.ecma-international.org`

fuzzing generates test inputs by randomly modifying an initial input. We used a Microsoft-internal widely-used fuzzing tool.

fuzzing+grammar generates test inputs by creating random strings from a given grammar. We used a strategy that generates strings of a given length uniformly at random [79], i.e., each string of a given length is equally likely.

concolic generates test inputs using the concolic testing algorithm of Section 2. We used SAGE [48], an existing tool that implements concolic testing for Windows programs.

concolic+tokens extends concolic testing with only the lexical part of the grammar, i.e., marks token identifiers as symbolic, instead of individual input bytes, but does not use a grammar. This strategy was implemented as an extension of SAGE.

concolic+grammar is our the main strategy, i.e., the grammar-based concolic testing. This strategy extends concolic testing with both symbolic tokens and an input grammar. This strategy was implemented as an extension of SAGE.

Figure 4-4 tabulates the strategies used in the evaluation and shows their characteristics.

strategy	seed inputs	random	tokens
fuzzing	✓	✓	
fuzzing+grammar		✓	✓
concolic	✓		
concolic+tokens	✓		✓
concolic+grammar	✓		✓

Figure 4-4: Test input generation strategies evaluated and their characteristics. The seed inputs column indicates which strategies require initial seed inputs from which to generate new inputs. The random column indicates which strategies use randomization. The tokens column indicates which strategies use the lexical specification (i.e., tokens) of the input language. Each technique's name indicates whether the technique uses a grammar and whether is it concolic testing or fuzzing.

Other strategies are conceivable. For example, concolic testing could be combined directly with the grammar, without tokens. Doing so requires transforming the grammar into a scannerless grammar [99]. Another possible strategy is bounded exhaustive enumeration [72, 108]. We have not included the latter in our evaluation because, while all other strategies we evaluated can be time-bounded (i.e., can be stopped at any time), exhaustive enumeration up to some input length is biased if terminated before completion, which makes it hard to fairly compare to time-bounded techniques.

4.5.3 Methodology

We used randomly generated seed inputs. To avoid bias when using test generation strategies that require seed inputs (see Figure 4-4), we used 50 seed inputs with 15 to 20 tokens generated randomly from the grammar. Section 4.5.4 provides more information about selecting the size of seed inputs. Also, to avoid bias across all strategies, we ran all experiments inside the same test harness.

The *concolic+tokens* and *concolic+grammar* strategies require identifying the tokenization function that creates grammar tokens. Our implementation allows doing so in a simple way, by overriding a single function in our framework.

For each of the examined modules (lexer, parser, and code generator), we measured the reachability rate, i.e., the percentage of inputs that execute at least one instruction of the module. Deeper modules always have lower reachability rates.

We measured instruction coverage, i.e., the ratio of the number of unique executed instructions to all instructions in the module of interest. This coverage metric seems the most suitable for our needs, since we want to estimate the error-finding potential of the generated inputs, and blocks with more instructions are more likely to contain errors than short blocks. In addition to the total instruction coverage for the interpreter, we also measured coverage in the lexer, parser, and code generator modules.

We ran each test input generation strategy for 2 hours. The 2-hour time included *all* experimental tasks: program execution, symbolic execution (where applicable), constraint solving (where applicable), generation of new inputs, and coverage measurements.

For reference, we also include coverage data and reachability results obtained with a "manual" test suite, created over several years by the developers and testers of this JavaScript interpreter. The suite consists of more than 2,800 hand-crafted inputs that exercise the interpreter thoroughly.

As an additional control, we used a test suite with *full production coverage*, created by using Purdom's algorithm [74, 94], which gives a set of strings that cover all grammar productions.

4.5.4 Seed Size Selection

Four of our generation strategies require seed inputs (Figure 4-4). To avoid bias stemming from using arbitrary inputs, we used inputs generated randomly from the JavaScript grammar. The length of the seed inputs may influence subsequent test input generation. To select the right length, we generated inputs of different sizes and measured the coverage achieved by each of those inputs as well as what percentage of inputs reaches the code generator. For each length, we generated 100 inputs and performed the measurements only for those inputs.

The findings (Figure 4-5) are not immediately intuitive: longer inputs achieve, on average, lower total coverage. The reason is that the official JavaScript grammar is only a partial specification of what constitutes syntactic validity. The grammar describes an over-approximation of the set of inputs acceptable by the parser.

size (tokens)	reach code gen. %	average coverage %	maximum coverage %
6	100	9	9
10	76	8	9
20	67	8	10
30	38	8	10
50	9	7	10
100	1	6	10
120	0	6	7
150	0	6	7
200	0	6	7

Figure 4-5: Coverage statistics for nine sets of 100 inputs each, generated randomly from the JavaScript grammar using the same uniform generator as *fuzzing+grammar*. The reach code gen. column displays the percentage of the generated inputs that reach the code generator module. The two right-most columns display the average and the maximum coverage of the whole interpreter for the generated inputs.

Longer, randomly generated, inputs are more likely to be accepted by the grammar and *rejected* by the parser. For example, the grammar specifies that `break` statements may occur anywhere in the function body, while the parser enforces that `break` statements may appear only in loops and `switch` statements. Enforcing this is possible by modifying the grammar but it would make the grammar much larger. Another example of over-approximation concerns line breaks and semicolons. The standard specifies that certain semicolons may be omitted, as long as there are appropriate line breaks in the file[3]. However, the grammar does not enforce this requirement and allows omitting all semicolons.

We selected 15 to 20 as the size range, in tokens, of the input seeds we use in other experiments. The results in Figure 4-5 indicate that this length makes the seed inputs variable without sacrificing the reachability rate (i.e., reachability of the code generation module).

4.5.5 Results

Coverage and Reachability

Figure 4-6 tabulates the coverage and reachability results for the 2-hour runs with each of the five automated test generation strategies previously discussed. For comparison, results obtained with the manually-written test suite are also included, even though running it requires between 2 and 3 hours (the 2,820 input JavaScript programs are typically much larger).

[3] See Section 7.9 of the specification: `http://interglacial.com/javascript_spec/a-7.html#a-7.9`

strategy	inputs	coverage %				reachability %		
		total	lexer	parser	gen.	lexer	parser	gen.
fuzzing	8658	14	25	25	52	99	99	18
fuzzing+grammar	7837	12	22	24	61	100	100	72
concolic	6883	15	26	29	54	99	99	17
concolic+tokens	3086	16	35	39	53	100	100	16
concolic+grammar	2378	20	25	42	82	100	100	81
seed inputs	50	11	18	21	51	100	100	66
full prod. coverage	15	12	20	27	52	100	100	53
manual test suite	2820	59	62	76	92	100	100	100

Figure 4-6: Coverage and reachability results for 2-hour runs (in %). The *seed inputs* row lists statistics for the seed inputs used by some of the test generation strategies (see Sections 4.5.3 and 4.5.4). The *manual test suite* takes more than 2 hours (less than 3 hours) to run and is included here for reference. The inputs column gives the number of inputs tested by each strategy. The coverage columns give the instruction coverage for lexer, parser, and code generator (gen.) modules. The reachability columns give the percentage of inputs that reach the module's entrypoint.

Our main strategy, *concolic+grammar*, achieved the best total coverage, as well as the best coverage in the deepest examined module, the code generator. It achieved results that are closest to the manual test suite, which predictably provides the best coverage. The manual suite is diverse and extensive, but was developed at the cost of many man-months of work. In contrast, *concolic+grammar* requires minimal human effort, and quickly generates relatively good test inputs. We can also observe the following.

- *Grammar-based concolic* testing achieved much better coverage than regular *concolic* testing.
- *Grammar-based concolic* testing performed significantly better than grammar-based fuzzing (*fuzzing+grammar*). Even though the latter strategy achieved good coverage in the code generator, concolic strategies outperformed fuzzing ones in total coverage.
- *Grammar-based concolic* testing achieved the highest coverage using the fewest inputs, which means that this strategy generates inputs of higher quality.
- *Concolic* testing was not much better than simple *fuzzing*. When testing programs with highly-structured inputs, concolic testing, with the power of symbolic execution and constraint solving, did not improve much over simple fuzzing. Furthermore, in the code generator, neither strategy improved coverage much above the initial set of seed inputs.
- Full production coverage did not correspond to high code coverage. However, our results show that a combination of specification-based and implementation-based testing (such as our grammar-based concolic testing) can perform better than either approach alone.

- Almost all tested inputs reached the lexer. A few inputs generated by the *fuzzing* and *concolic* strategies contained invalid (e.g., non-ASCII) characters and the interpreter rejected them before using the lexer. To exercise the interpreter well, inputs must reach the deepest module, the code generator. The results show that *concolic+grammar* had the highest percentage of such deep-reaching inputs.

In summary, the results of these experiments validate our claim that grammar-based concolic testing is effective in reaching deeper into the tested application and exercising the code more thoroughly than other automated test-generation strategies.

Relative Coverage

Grammar-based concolic testing creates test input that cover code not covered by other strategies. Figure 4-7 compares the instructions covered with *concolic+grammar* and the other analyzed strategies. The numbers show that the inputs generated by *concolic+grammar* covered most of the instructions covered by the inputs generated by the other strategies (see the small numbers in the $S - GBC$ column), while covering many other instructions (see the relatively larger numbers in the $GBC - S$ column).

strategy S	$S - GBC$	$GBC \cap S$	$GBC - S$
fuzzing	1	13	7
fuzzing+grammar	0	12	9
concolic	2	13	7
concolic+tokens	2	14	6

Figure 4-7: Relative coverage in % compared to *concolic+grammar* (*GBC*). The column $S - GBC$ gives the percentage of instructions covered by each strategy but *not* by *GBC*. The column $GBC \cap S$ gives the percentage of instructions covered by both strategies. The last column gives the percentage of instructions covered by *GBC* and not by S.

Combined with the results of Section 4.5.5, this shows that *concolic+grammar* achieved the highest total coverage, highest reachability rate, and highest coverage in the deepest module, while using the smallest number of inputs.

Context-Free Constraint Solver Performance

Grammar-based constraint solver was efficient. To measure the performance of the solver, we repeated the 2-hour *concolic+grammar* run 9 times with different sizes of seed inputs (between 10 and 200 tokens). The average number of solver calls per symbolic execution was between 23 and 53 (with no obvious correlation between seed input size and the average number of solver calls). The results were that up to 58% of total execution time was spent in the constraint solver (with no obvious correlation between seed size and solving time). Such solving times are typical in concolic testing (e.g., a recent report [47] indicates up to 62% of the test generation time spent in the constraint solver).

Grammar-based Search Tree Pruning

Grammar-based concolic testing was effective in identifying dead-end inputs. In our 2-hour experiments, 29% of grammar constraints were unsatisfiable. When a grammar constraint is unsatisfiable, the corresponding search tree is pruned because there is no input that satisfies the constraint and is valid according to the grammar.

Statistics on Concolic Executions

strategy	created constraints %			symbolic	avg.	avg.
	lexer	parser	code gen.	execs	symb. vars	constraints
concolic	67	33	0	131	57	298
concolic+tokens	0	98	2	170	12	67
concolic+grammar	0	98	2	143	21	113

Figure 4-8: Concolic execution statistics for 2-hour runs of concolic strategies. The created constraints columns shows the percentages of all symbolic constraints created in the three analyzed modules of the JavaScript interpreter. The symbolic execs column gives the total number of symbolic executions during each run. The two right-most columns give the average number of symbolic variables per symbolic execution and the average number of symbolic constraints per symbolic execution.

Figure 4-8 presents statistics related to the concolic executions performed during the 2-hour runs of each of the three concolic strategies evaluated. We make the following observations.

- All three concolic strategies performed roughly the same number of symbolic executions.
- However, the *concolic* strategy created a larger average number of symbolic variables because it operated on characters, while the other two strategies worked on tokens (cf. Figure 4-4).
- The *concolic+tokens* strategy created the smallest average number of symbolic variables per execution. This is because *concolic+tokens* generated many unparsable inputs (cf. Figure 4-6), which the parser rejected early and therefore no symbolic variables were created for the tokens after the parse error.

Token-based strategies avoided path explosion in the lexer. Figure 4-8 shows how constraint creation was distributed among the lexer, parser, and code generator modules of the JavaScript interpreter. The two token-based strategies (*concolic+tokens* and *concolic+grammar*) generated no constraints in the lexer. This helped to avoid path explosion in that module. Those strategies did explore the lexer (indeed, Figure 4-6 shows high coverage) but they did not get lost in the lexer's error-handling paths.

All strategies created constraints in the deepest, code generator, module. However, there were few such constraints because the parser transforms the stream of tokens into an Abstract Syntax Tree (AST) and subsequent code, such as the code generator, operates on the AST. When processing the AST in later stages, symbolic variables associated with input bytes or tokens are largely absent, so symbolic execution does not create constraints from code branches in these stages. The number of symbolic constraints in those deeper stages could be increased by associating symbolic variables with other values returned by the tokenization function such as string and integer values associated with some tokens.

4.6 Related Work

Concolic testing [18, 46, 102] finds errors without generating false alarms and requires no domain knowledge. Our work enhances concolic testing by taking advantage of a formal grammar representing valid inputs, thus helping the generation of test inputs that exercise longer execution paths.

Miller's pioneering fuzzing tool [81] generated streams of random bytes, but most popular fuzzers today support some form of grammar representation, e.g., SPIKE[4], Peach[5], FileFuzz[6], Autodafé[7]. Sutton et al. [109] present a survey of fuzzing techniques and tools. Work on grammar-based test input generation started in the 1970s [53, 94] and can be broadly divided into random [26, 74, 76, 106] and exhaustive generation [69,72]. Imperative generation [21,30,89] is a related approach in which a custom-made program generates the inputs (in effect, the program encodes the grammar). In systematic approaches, test inputs are created from a specification, given either a special piece of code (e.g., Korat [14]) or a logic formula (e.g., TestEra [62]). Grammar-based test input generation is an example of model-based testing (see Utting et al. for a survey [112]), which focuses on covering the specification (model) when generating test inputs to check conformance of the program with respect to the model. Our work also uses formal grammars as specifications. However, in contrast to fuzzing approaches, our approach analyses the code of the program under test and derives new test inputs from it.

Path explosion in concolic testing can be alleviated by performing test generation compositionally [44], by testing functions systematically in isolation, encoding and memoizing test results as function summaries using function input preconditions and output postconditions, and re-using such summaries when testing higher-level functions. A grammar can be viewed as a special form of user-

[4] `http://www.immunitysec.com/resources-freesoftware.shtml`

[5] `http://peachfuzz.sourceforge.net/`

[6] `http://labs.idefense.com/software/fuzzing.php`

[7] `http://autodafe.sourceforge.net`

provided compact "summary" for the entire lexer/parser, that may include over-approximations. Computing such a finite-size summary automatically may be impossible due to infinitely many paths or limited symbolic reasoning capability when analyzing the lexer/parser. Grammar-based concolic testing and test summaries are complementary techniques which could be used simultaneously.

Another approach to path explosion consists of abstracting lower-level functions using software stubs, marking their return values as symbolic, and then refining these abstractions to eliminate infeasible program paths [71]. In contrast, grammar-based concolic testing is always grounded in concrete executions, and thus does not require the expensive step of removing infeasible paths.

Emmi et al. [36] extend systematic testing with constraints that describe the state of the data for database applications. Our approach also solves path and data constraints simultaneously, but ours is designed for compilers and interpreters instead of database applications.

Majumdar and Xu's recent and independent work [72] is closest to ours. These authors combine grammar-based fuzzing with concolic testing by exhaustively pre-generating strings from the grammar (up to a given length), and then performing concolic testing starting from those pre-generated strings, treating only variable names, number literals etc. as symbolic. Exhaustive generation inhibits scalability of this approach beyond very short inputs. Also, the exhaustive grammar-based generation and the concolic testing parts do not interact with each other in Majumdar and Xu's framework. In contrast, our grammar-based concolic testing approach is more powerful as it exploits the grammar for solving constraints generated during symbolic execution to generate input variants that are guaranteed to be valid.

4.7 Conclusion

We introduced grammar-based concolic testing to enhance the effectiveness of concolic testing for applications with complex, highly-structured inputs, such as interpreters and compilers. Grammar-based concolic testing tightly integrates implementation-based and specification-based testing, and leverages the strengths of both.

As shown by our case studies, grammar-based concolic testing generates tests that exercise more code in the deeper, harder-to-test layers of the program under test. In our experiments with UNIX programs, our technique lead to up to 2× improvements in line coverage, eliminated all illegal inputs, and enabled discovering 3 distinct, previously unknown, inputs that led to infinitely-looping program execution. In our experiments with the JavaScript interpreter in Internet Explorer 7, grammar-based concolic testing strongly outperformed both concolic testing and fuzzing. Code generator coverage improved from 61% to 81% and deep reachability improved from 72% to 80%. Deep parts of the application are the hardest to test automatically and our technique shows how to address this.

Since grammars are often partial specifications of valid inputs, grammar-based fuzzing approaches are fundamentally limited. Thanks to concolic testing, some of this incompleteness can be recovered, which explains why grammar-based concolic testing also outperformed grammar-based fuzzing in our experiments.

Chapter 5

Concolic Security Testing

Concolic security testing finds security vulnerabilities in Web applications. Multi-user Web applications are responsible for handling much of the business on the Internet. Such applications often manage sensitive data for many users, and that makes them attractive targets for attackers: up to 70% of recently reported vulnerabilities affected Web applications [19]. Therefore, security and privacy are of great importance for Web applications.

Two classes of attacks are particularly common and damaging [19]. In SQL injection (SQLI), the attacker executes malicious database statements by exploiting inadequate validation of data flowing from the user to the database. In cross-site scripting (XSS), the attacker executes malicious code on the victim's machine by exploiting inadequate validation of data flowing to statements that output HTML.

Previous approaches to identifying SQLI and XSS vulnerabilities and preventing exploits include defensive coding, static analysis, dynamic monitoring, and test generation. Each of these approaches has its own merits, but also shortcomings. Defensive coding [25] is error-prone and requires rewriting existing software to use safe libraries. Static analysis tools [70, 115] can produce false warnings and do not create concrete examples of inputs that exploit the vulnerabilities. Dynamic monitoring tools [52, 92, 107] incur runtime overhead on the running application and do not detect vulnerabilities until the code has been deployed. Black-box test generation does not take advantage of the application's internals, while previous white-box techniques [116] have not been shown to discover unknown vulnerabilities.

We present concolic security testing, a novel technique for identifying SQLI and XSS vulnerabilities. Unlike previous approaches, our technique works on unmodified existing code, creates concrete inputs that expose vulnerabilities, operates before software is deployed, has no overhead for the released software, and analyzes application internals to discover vulnerable code. As an implementation of our technique, we created ARDILLA, an automated tool for creating SQLI and XSS attacks in PHP/MySQL applications. ARDILLA is a white-box testing tool, i.e., it analyzes the source code of the application. ARDILLA is designed for testing PHP applications before deployment. Security vulnerabilities that ARDILLA identifies can be fixed before the software reaches the users because ARDILLA creates concrete attacks that exploit the vulnerability. In our experiments, ARDILLA discovered 68 previously unknown vulnerabilities in five applications.

Concolic security testing has five components: input generation, concolic execution, concolic database, attack-candidate generation, and attack-candidate checking. We now briefly discuss these components.

Concolic security testing can use any **input generator**. ARDILLA uses an input generator that is based on concolic testing [2]. During each execution, this input generator monitors the program to record path constraints that capture the outcome of control-flow predicates. The input generator automatically and iteratively generates new inputs by negating one of the observed constraints and solving the modified constraint system. Each newly-created input aim to explore an additional execution path.

ARDILLA's vulnerability detection is based on **concolic execution**. ARDILLA's concolic execution significantly enhances that used in the input generator. ARDILLA tracks symbolic values for all concrete values in the program. In particular, ARDILLA maintains symbolic string expressions for all string runtime values. ARDILLA marks data coming from the user as symbolic, tracks the flow of user data in the program, and checks whether user-derived data can reach *sensitive sinks*. An example of a sensitive sink is the PHP `mysql_query` function, which executes a string argument as a MySQL statement. If a string derived from user data is passed into this function, then an attacker can potentially perform an SQL injection if the user-derived string affects the structure of the SQL query. Similarly, passing user-derived data into functions that output HTML can lead to XSS attacks.

ARDILLA's concolic execution is unique in that it tracks the flow of symbolic information through the database, using a **concolic database**. When the concrete values are stored in the database, the symbolic information is stored with them. When the values are later retrieved from the database, they are marked with the stored symbolic values. Thus, only data that was marked as symbolic upon storing is marked as symbolic upon retrieval. This precision makes ARDILLA able to accurately detect second-order (persistent) XSS attacks. By contrast, previous techniques either treat *all* data retrieved from the database as user-derived [114,115] (which may lead to false warnings) or treat all such data as not user-derived [70] (which may lead to missing real vulnerabilities).

To convincingly demonstrate a vulnerability, a tester or a tool must create concrete attack vectors [9,37,43]. ARDILLA can **create attack candidates** in two ways: using a library of known attack patterns, or using a string-constraint solver, such as HAMPI (Chapter 3). This step is necessary because not every flow of user data to a sensitive sink indicates a vulnerability because the data may flow through routines that check or sanitize it.

ARDILLA then **checks the attack candidate** by analyzing the difference between the parse trees of application outputs (SQL and HTML). ARDILLA checks whether the candidate may subvert the behavior of a database or a Web browser, respectively. This step enables ARDILLA to reduce the number of false warnings and to precisely identify real vulnerabilities.

Contributions and Results

- Concolic security testing, a fully-automatic technique for creating SQLI and XSS attack vectors, including those for second-order (persistent) XSS attacks (Section 5.2)
- A novel technique that determines whether the flow of user data to a sensitive sink is a vulnerability, using input mutation and output comparison (Sections 5.3.3 and 5.3.4).
- A novel approach to symbolically tracking the flow of user data through a database (Section 5.3.5).
- ARDILLA, a tool that implements the technique for PHP (Section 5.3).
- Evaluation of ARDILLA on five PHP applications (Section 5.4). We found 68 previously-unknown vulnerabilities (23 SQLI, 33 first-order XSS, and 12 second-order XSS).

We describe the problem of attacks on Web-applications (Section 5.1), present the concolic security testing technique (Section 5.2), the ARDILLA tool (Section 5.3), and an experimental evaluation of the tool (Section 5.4). We finish with related work (Section 5.5).

5.1 SQL Injection and Cross-Site Scripting Attacks

This section describes SQLI and XSS Web-application vulnerabilities and illustrates attacks that exploit them.

SQL Injection

A SQLI vulnerability results from the application's use of user input in constructing database statements. The attacker invokes the application, passing as an input a (partial) SQL statement, which the application executes. The attack permits the attacker to get unauthorized access to, or to damage, the data stored in a database. To prevent this attack, applications need to sanitize input values that are used in constructing SQL statements, or else reject potentially dangerous inputs.

First-order XSS

A first-order XSS (also known as Type 1, or reflected, XSS) vulnerability results from the application inserting part of the user's input in the next HTML page that it renders. The attacker uses social engineering to convince a victim to click on a (disguised) URL that contains malicious HTML/JavaScript code. The victim's browser then displays HTML and executes JavaScript that was part of the attacker-crafted malicious URL. The attack can result in website defacement, stealing of browser cookies and other sensitive user data. To prevent first-order XSS attacks, users need to check link anchors before clicking on them, and applications need to reject or modify input values that may contain script code.

Second-order XSS

A second-order XSS (also known as Type 2, persistent, or stored, XSS) vulnerability results from the application storing (part of) the attacker's input in a database, and later inserting it in an HTML page that is displayed to multiple victim users (e.g., in an online bulletin-board application).

It is harder to prevent second-order XSS than first-order XSS. Applications need to reject or sanitize input values that may contain script code and are either displayed in HTML output, *or* used in database commands.

Second-order XSS is much more damaging than first-order XSS because: (i) social engineering is not required (the attacker can directly supply the malicious input without tricking users into clicking on a URL), and (ii) a single malicious script planted once into a database executes on the browsers of many victim users.

5.1.1 Example PHP/MySQL Application

PHP is a server-side scripting language widely used in creating Web applications. The program in Figure 5-1 implements a simple message board that allows users to read and post messages, which are stored in a MySQL database. To use the message board, users of the program fill an HTML form (not shown here) that communicates the inputs to the server via a specially formatted URL, e.g.,

```
http://www.mysite.com/?mode=display&topicid=1
```

In this example URL, the input has two key-value pairs: `mode=display` and `topicid=1`. The input passed inside the URL is available to the PHP program via the `$_GET` associative array.

The program can operate in two modes: posting a message or displaying all messages for a given topic. When posting a message, the program constructs and submits the SQL statement to store the message in the database (lines 25 and 28) and then displays a confirmation message (line 29). In the displaying mode, the program retrieves and displays messages for the given topic (lines 39, 40, and 44).

This program is vulnerable to the following attacks, all of which our technique can automatically generate:

SQL Injection Attack

Both database queries, in lines 28 and 40, are vulnerable but we discuss only the latter, which exploits the lack of input validation for `topicid`.

Consider the following string passed as the value for input parameter `topicid`:

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

```
// exit if parameter ’mode’ is not provided
if(!isset($_GET[’mode’])){
  exit;
}

if($_GET[’mode’] == "add")
  addMessageForTopic();
else if($_GET[’mode’] == "display")
  displayAllMessagesForTopic();
else
  exit;

function addMessageForTopic(){
  if(!isset($_GET[’msg’]) ||
     !isset($_GET[’topicid’]) ||
     !isset($_GET[’poster’])){
        exit;
  }

  $my_msg = $_GET[’msg’];
  $my_topicid = $_GET[’topicid’];
  $my_poster = $_GET[’poster’];

  //construct SQL statement
  $sqlstmt = "INSERT INTO messages VALUES(’$my_msg’,’$my_topicid’)";

  //store message in database
  $result = mysql_query($sqlstmt);
  echo "Thank you $my_poster for using the message board";
}

function displayAllMessagesForTopic(){
  if(!isset($_GET[’topicid’])){
    exit;
  }

  $my_topicid = $_GET[’topicid’];

  $sqlstmt = "SELECT msg FROM messages WHERE topicid=’$my_topicid’";
  $result = mysql_query($sqlstmt);

  //display all messages
  while($row = mysql_fetch_assoc($result)){
    echo "Message " . $row[’msg’];
  }
}
```

Figure 5-1: Example PHP program that implements a simple message board using a MySQL database. This program is vulnerable to SQL injection and cross-site scripting attacks. Section 5.1.1 discusses the vulnerabilities. (For simplicity, the figure omits code that establishes a connection with the database.)

This string leads to an attack because the query that the program submits to the database in line 40,

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

contains a tautology in the `WHERE` clause and will retrieve all messages, possibly leaking private information.

To exploit the vulnerability, the attacker must create an *attack vector*, i.e., the full set of inputs that make the program follow the exact path to the vulnerable `mysql_query` call and execute the attack query. In our example, the attack vector must contain at least parameters `mode` and `topicid` set to appropriate values. For example:

```
mode    → display
topicid → 1' OR '1'='1
```

First-order XSS Attack

This attack exploits the lack of validation of the input parameter `poster`. After storing a message, the program displays a confirmation note (line 29) using the local variable `my_poster`, whose value is derived directly from the input parameter `poster`. Here is an attack vector that, when executed, opens a popup window on the user's computer:

```
mode    → add
topicid → 1
msg     → Hello
poster  → Villain<script>alert("XSS")</script>
```

This particular popup is innocuous; however, it demonstrates the attacker's ability to execute script code in the victim's browser (with access to the victim's session data and permissions). A real attack might, for example, send the victim's browser credentials to the attacker.

Second-order XSS Attack

This attack exploits the lack of SQL validation of parameter `msg` when storing messages in the database (line 25) and the lack of HTML validation when displaying messages (line 44). The attacker can use the following attack vector to store the malicious script in the application's database.

```
mode    → add
topicid → 1
msg     → Hello<script>alert("XSS")</script>
poster  → Villain
```

Now *every* user whose browser displays messages in topic `1` gets an unwanted popup. For example, executing the following innocuous input results in an attack:

```
mode    → display
topicid → 1
```

5.2 Technique

Our technique generates a set of concrete inputs, executes the program under test with each input, and dynamically observes whether and how data flows from a user input to a sensitive sink (e.g., a function such as `mysql_query` or `echo`), including any data flows that pass through a database. If an input reaches a sensitive sink, our technique modifies the input by either using a string-constraint solver, or using a library of attack patterns, in an attempt to pass malicious data through the program.

This section first shows the five components of our technique (Section 5.2.1) and then describes the algorithms for automatically generating first-order (Section 5.2.2) and second-order (Section 5.2.3) attacks.

5.2.1 Technique Components

Figure 5-2 shows the architecture of our technique and of the ARDILLA tool that we created as an implementation of the technique for PHP. Here, we briefly describe its four components as an aid in understanding the algorithms. Section 5.3 describes ARDILLA and the four components in detail.

- The **Input Generator** creates a set of inputs for the program under test, aiming to cover many execution paths.

- The **Concolic Executor** runs the program on each input produced by the input generator and tracks how the user input flows into sensitive sinks. For each sensitive sink, the executor outputs a set of symbolic expressions on string variables (input parameters). These expressions indicate how the input values flow into the sink.

- The **Attack Generator** takes a list of symbolic expressions (one for each sensitive sink), creates candidate attacks by modifying the inputs using a **String-Constraint Solver**.

- The **Attack Checker** runs the program on the candidate attacks to determine which are real attacks.

- The **Concolic Database** is a relational database engine that can execute SQL statements both concretely and symbolically. Our technique uses this component to track the flow of symbolic data through the database, which is critical for accurate detection of second-order XSS attacks.

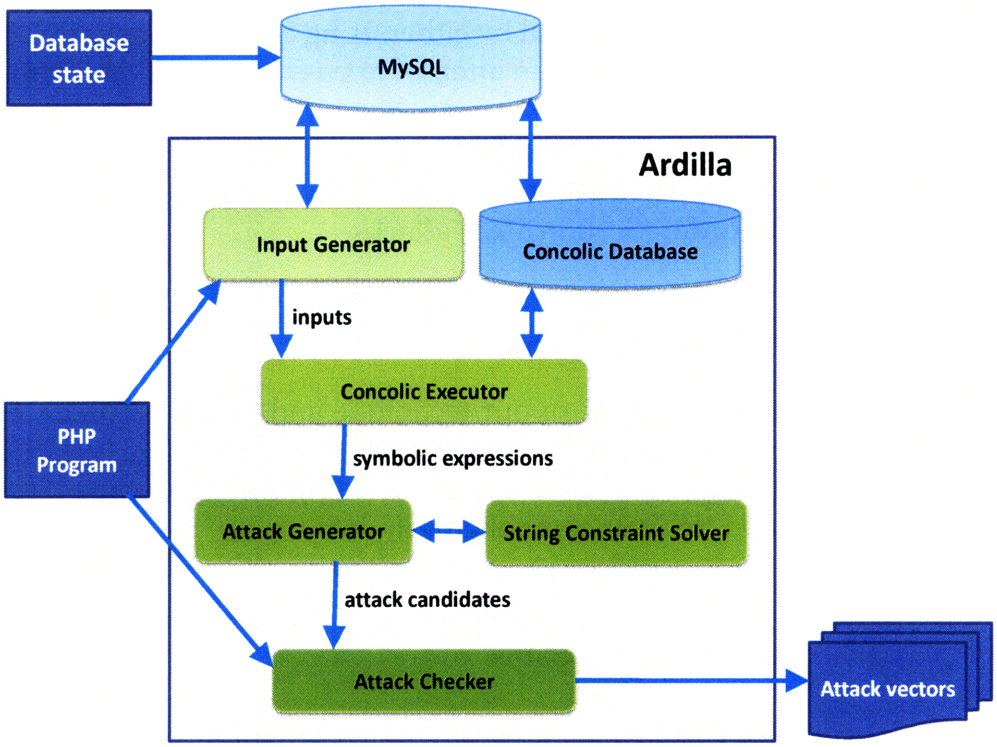


Figure 5-2: The architecture of ARDILLA. The inputs to ARDILLA are the PHP program and its associated database state. The output is a set of attack vectors for the program. Each attack vector is a complete input that exposes a security vulnerability.

5.2.2 First-order Attacks

Figure 5-3 shows the algorithm for generating SQLI and first-order XSS attacks (both called *first-order* because they do not involve storing malicious inputs in the database). The algorithms for creating SQLI and first-order XSS attacks are identical except for the sensitive sinks (`mysql_query` for SQLI, `echo` and `print` for XSS) and details in the attack generator/checker.

```
  parameters: program $\mathcal{P}$, database state $db$
  result    : SQLI or first-order XSS attack vectors
1 $attacks := \emptyset$;
2 while not $timeExpired()$ do
3   $input := generateNewInput(\mathcal{P})$;
4   $\langle symbExprs, db' \rangle := concolicExecution(\mathcal{P}, input, db)$;
5   $candidates := generateAttacks(symbExprs, \mathcal{P}, input)$;
6   $attacks := attacks \cup checkAttacks(candidates, \mathcal{P}, input)$;
7 return $attacks$;
```

Figure 5-3: Algorithm for creating SQLI and first-order XSS attacks.

The algorithm takes the program $\mathcal{P}$ under test and its associated database *db* populated with the proper tables and initial data (usually done via an installation script or taken from an existing installation). Until a time limit is reached, the algorithm generates new concrete inputs (line 3), runs the program on each input and collects symbolic expressions (line 4), generates candidate attacks vectors (line 5) and checks the candidates to find real attack vectors (line 6).

Example SQLI

Here is how our technique generates the SQL injection attack presented in Section 5.1.1. First, new inputs are successively generated and the program executes concolically on each input until some input allows the program to reach line 40 in the code in Figure 5-1, which contains the sensitive sink `mysql_query`. An example of such an input *I* is (our input generator picks `1` as the default "don't care" value):

```
mode    → display
topicid → 1
```

Second, the concolic executor runs the program on *I*, marks each input parameter as symbolic variable, and creates symbolic expressions for string values that flow into sensitive sinks. In the example, the concrete string value that reaches the sensitive sink is:

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

The corresponding symbolic expression is:

concat`("SELECT msg FROM messages WHERE topicid='",topicid,"'")` Third, ARDILLA use the HAMPI string-constraint solver to find a value of the parameter `topicid` for which the query is an attack. ARDILLA uses a list of SQLI attack patterns, e.g., `OR '1'='1'`. The string-constraint solver finds a value for which the concrete value is a valid SQL statement *and* contains the attack pattern (Chapter 3 describes the necessary HAMPI string constraints.) One such value is `1' OR '1'='1`. Using this value alters input *I* into *I*′:

```
mode    → display
topicid → 1' OR '1'='1
```

Fourth, the attack checker runs the program on *I*′ and determines that *I*′ is a real attack, i.e., changes the syntactic structure of the SQL statement.

Finally, the algorithm outputs *I*′ as an attack vector for the first-order SQLI vulnerability in line 29 of Figure 5-1.

Example XSS

Here is how our technique generates the first-order XSS attack presented in Section 5.1.1. First, new inputs are successively generated and the program executes concolically on each input until some input allows the program to reach line 29 in the code in Figure 5-1, which contains the sensitive sink `echo`. An example of such an input *I* is:

```
mode    → add
topicid → 1
msg     → 1
poster  → 1
```

(Even though only the value of `mode` determines whether execution reaches line 29, all parameters are required to be set; otherwise the program rejects the input in line 17.)

Second, the concolic executor runs the program on *I*, marks each input parameter as symbolic variable, and creates symbolic expressions for string values that flow into sensitive sinks. In the example, the executor determines that the value of the parameter `poster` flows into the local variable `my_poster`, which flows into the sensitive sink `echo` in line 29:

```
$my_poster = $_GET['poster'];
...
echo "Thank you $my_poster for using the message board";
```

The corresponding symbolic expression is:

concat(`"Thank you ",poster," for using the message board"`)

Third, the attack generator mutates the input *I* by replacing the value of all parameters in the symbolic expressions (here only `poster`) with XSS attack patterns. An example pattern is `<script>alert("XSS")</script>`. Picking this pattern alters input *I* into *I*′:

```
mode     → add
topicid  → 1
msg      → 1
poster   → <script>alert("XSS")</script>
```

Fourth, the attack checker runs the program on I' and determines that I' is a real attack, i.e., changes the set of script-inducing elements in the HTML output of the program.

Finally, the algorithm outputs I' as an attack vector for the first-order XSS vulnerability in line 29 of Figure 5-1.

5.2.3 Second-order Attacks

Figure 5-4 shows the algorithm for generating second-order XSS attacks, which differs from the first-order algorithm by using a concolic database and by running the program on two inputs during each iteration. The first input represents one provided by an attacker, which contains malicious values. The second input represents one provided by a victim, which does not contain malicious values. The algorithm tracks the flow of data from the attacker's input, through the database, and to a sensitive sink in the execution on the victim's innocuous input.

```
parameters: program $\mathcal{P}$, database state $db$
result    : second-order XSS attack vectors
1  $inputs := \varnothing$;
2  $attacks := \varnothing$;
3  $db_{con} := makeSymbolicCopy(db)$;
4  while $not\ timeExpired()$ do
5    $inputs := inputs \cup generateNewInput(\mathcal{P})$;
6    $input_1 := pickInput(inputs)$;
7    $input_2 := pickInput(inputs)$;
8    $\langle symbExprs_1, db'_{con} \rangle := concolicExecution(\mathcal{P}, input_1, db_{con})$;
9    $\langle symbExprs_2, db''_{con} \rangle := concolicExecution(\mathcal{P}, input_2, db'_{con})$;
10   $candidates := attacks \cup generateAttacks(symbExprs_2, \mathcal{P}, \langle input_1, input_2 \rangle)$;
11   $attacks := attacks \cup checkAttacks(candidates, \mathcal{P}, \langle input_1, input_2 \rangle)$;
12 return $attacks$;
```

Figure 5-4: Algorithm for creating second-order XSS attacks.

The algorithm takes the program $\mathcal{P}$ under test and a database db. In the first step (line 3), the algorithm makes a symbolic copy of the concrete database, thus creating a concolic database. Then, until a time limit expires, the algorithm generates new concrete inputs and attempts to create attack vectors by modifying the inputs. The algorithm maintains a set of inputs generated so far (in the *inputs* variable), from which, in each iteration, the algorithm picks two inputs (lines 6 and 7). Then, the algorithm performs concolic executions of the two inputs, in sequence, (lines 8 and 9) using the concolic database. The first execution (simulating the attacker) sets the state of the database (db'_{con}) that the second execution (simulating the victim) uses. Next, the attack generator (line 10) creates candidate second-order XSS attack scenarios (i.e., input pairs) using a library of attack patterns. Finally, the attack checker (line 11) checks candidate second-order XSS attack scenarios (i.e., input pairs).

To favor execution paths that lead to second-order XSS attacks, on line 6 our implementation heuristically picks an input that executes a database write, and on line 7 picks an input that executes a database read on the same table.

Example XSS

Here is how our technique generates the second-order XSS attack introduced in Section 5.1.1. First, the input generator creates inputs and picks the following pair I_1:

```
mode    → add
topicid → 1
msg     → 1
poster  → 1
```

and I_2:

```
mode    → display
topicid → 1
```

Second, the concolic executor runs the program on I_1, using the concolic database. During this execution, the program stores the value 1 of the input parameter msg (together with the symbolic expression msg) in the database (line 25 of Figure 5-1).

Third, the concolic executor runs the program on I_2, using the concolic database. During this execution, the program retrieves the value 1 from the database (together with the value's stored symbolic expression msg) and outputs the value via the echo in line 44. echo is a sensitive sink, and the symbolic expression associated with the string value that flows into the sink is *concat*("Thank you ", msg), where msg is a parameter from I_1. Thus, the algorithm has dynamically tracked the flow of user data across two executions: from msg to the local variable my_msg (line 20), *into the database* (line 28), back out of the database (line 40), into the $row array (line 43), and finally as a parameter to echo (line 44).

Fourth, the attack generator uses the attack-pattern library (similarly to the first-order XSS example before) to alter msg in I_1 to create an attack-candidate input I'_1:

```
mode    → add
topicid → 1
msg     → <script>alert("XSS")</script>
poster  → 1
```

Fifth, the attack checker runs the program, in sequence, on I'_1 and I_2 (note that I_2

remains unchanged), and determines that this sequence of inputs is an attack scenario.

Finally, the algorithm outputs the pair $\langle I_1', I_2 \rangle$ as a second-order XSS attack scenario that exploits the vulnerability in line 44 of Figure 5-1.

5.3 The ARDILLA Tool for Creating SQLI and XSS Attacks

ARDILLA is an implementation of concolic security testing. ARDILLA generates concrete attack vectors for Web applications written in PHP. The user of ARDILLA needs to specify the type of attack (SQLI, first-order XSS, or second-order XSS), the PHP program to analyze, and the initial database state. The outputs of ARDILLA are attack vectors. This section describes ARDILLA's implementation of each component of the technique described in Section 5.2.

5.3.1 Dynamic Input Generator

The dynamic input generator creates inputs for the PHP program under test. Inputs for PHP Web applications are Web server requests: their parameters are mappings from keys (strings) to values (strings and integers) in associative arrays such as `$_GET` and `$_POST`.

ARDILLA uses the input-generation component from Apollo [2], but ARDILLA could potentially use any generator for PHP applications such as the one described by Wassermann et al. [116]. The Apollo input generator is based on concolic testing (Chapter 2). Here, we briefly describe Apollo's technique, which ARDILLA uses as a black box.

For each program input (starting with an arbitrary well-formed concrete input, and then using subsequently-generated ones), the input generator executes the program concretely and also collects symbolic constraints for each runtime value. These constraints describe an input that follows a given execution path through the program. Negating the symbolic constraint at a branch point (e.g., an `if` statement) gives a set of constraints for a different path through the program. The input generator then attempts to solve those constraints to create a concrete input that executes the new path. The input generator repeats this process for each branch point in an execution, possibly generating many new inputs from each executed one.

5.3.2 Concolic Executor

The concolic executor runs the program under test on each input and tracks the dynamic data flow of input parameters throughout the execution. For each sensitive sink, the executor outputs the set of symbolic expressions for values that flow into the sink. ARDILLA's concolic execution is unique in that it can track the flow of symbolic data through the database, by using a concolic database (Section 5.3.5). Concolic execution in ARDILLA can be characterized by the following five components.

- **Symbolic variables** are derived from inputs (e.g., `$_GET` and `$_POST`). ARDILLA assigns a unique symbolic variable to each value read from an input parameter, identified by the value's origin. For example, ARDILLA assigns variable `msg` to a value retrieved from `$_GET['msg']`.

- **Symbolic expressions** describe how each runtime value is derived from symbolic variables. The grammar of ARDILLA's symbolic expressions contains symbolic variables, constants, string concatenation, and function calls (for built-in functions). For example, the symbolic expression *concat*(`"Thank you "`, `poster`, `" for posting"`) may correspond to a runtime value derived from input parameter `poster` via string concatenation. Another example: the symbolic expression `htmlentities`(*concat*(`"Hello "`, `poster`)) may correspond to a runtime value derived by calling a built-in function `htmlentities`.

- **Symbolic operations** specify how concrete runtime values acquire and lose associated symbolic expressions. ARDILLA propagates symbolic expressions unchanged across assignments and procedure calls in application code. At a call to a built-in PHP function (e.g., `chop`, which removes trailing whitespace from a string) that is not a **sanitizer** (see next component), ARDILLA constructs a symbolic expression for the return value that specifies the function's name and symbolic values of parameters. For string values created from concatenation, ARDILLA constructs concatenation symbolic expressions. At a call to a database function (e.g., `mysql_query`), ARDILLA stores or retrieves symbolic expressions for the concrete data values. (Section 5.3.5 describes the interaction of concolic execution with the database.)

- **Sanitizers** are built-in PHP functions that are known to sanitize inputs (i.e., modify the inputs to make them harmless for XSS or SQLI attacks). For example, `htmlentities` converts characters to HTML entities (e.g., < to `&lt;`) and makes the output safe from XSS attacks. At a call to a sanitizer function, ARDILLA creates an empty symbolic expression for the return value. A user of ARDILLA can optionally specify a list of sanitizers.

- **Sensitive sinks** are built-in PHP functions that are exploitable in XSS and SQLI attacks: for example, `echo` and `print` for XSS and `mysql_query` for SQLI. When reaching a call to a sensitive sink, ARDILLA records the symbolic expressions of the arguments, indicating a data flow from the inputs to the sink, and thus a possibility of an attack.

ARDILLA's concolic executor is implemented by modifying the Zend PHP interpreter[1] to perform regular program execution and to simultaneously perform concolic execution. This "shadow interpreter" approach enables using the modified interpreter for any program without source-code rewriting [2,16,48,60,110].

[1] `http://www.zend.com`

5.3.3 Attack Generator

The attack generator creates candidate attack vectors that are variants of the given input. The attack generator starts with an input for which there is dataflow from a parameter to a sensitive sink. For each parameter whose value flows into the sink (i.e., symbolic variable in the symbolic expression), the generator creates new inputs that differ only for that parameter. The generator can create new inputs in one of two modes: using a string-constraint solver or an attack-pattern library. When using a string-constraint solver, the generator finds the value of that parameter that, when used in an input, results in an attack string (a value that may result in an attack if supplied to a vulnerable input parameter). When using the attack-pattern library, the generator systematically replaces the value of that parameter by values taken from the library, i.e., a set of values that may result in an attack if supplied to a vulnerable input parameter.

Following previous work [41, 52], we define an SQLI attack as a syntactically valid (according to the grammar) SQL statement with a tautology in the `WHERE` clause, e.g., `OR 1=1`. In our experiments, we used 4 attack strings (tautology patterns), distilled from several security lists[2,3].

The input to HAMPI includes a partial SQL grammar (similar to that in Figure 3-2). 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. Each terminal is represented by a single unique character.

ARDILLA's XSS attack-pattern library[4] contains 113 XSS attack patterns, including many filter-evading patterns (that use various character encodings, or that avoid specific strings in patterns).

ARDILLA's goal is creating concrete exploits, not verifying the absence of vulnerabilities. Moreover, ARDILLA checks every candidate attack input. Therefore, ARDILLA is useful even given the pattern library's inevitable incompleteness (missing attack patterns), and potential unsoundness (patterns that do not lead to attacks).

The attack library needs to be integrated in ARDILLA to be effective; the library alone is not enough to construct attacks. ARDILLA constructs each attack input so that the execution reaches the vulnerable call site (using random values is ineffective for this purpose [2]). In particular, the constructed attack inputs contain many key-value pairs. Strings from the attack library constitute only 1 value in each attack input.

[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`

[3] ARDILLA's list omits attacks that transform one query into multiple queries, because the PHP `mysql_query` function only allows one query to be executed per call.

[4] `http://ha.ckers.org/xss.html`

5.3.4 Attack Checker

The attack checker determines whether a candidate input is an attack. The attack checker compares the candidate input's execution to that of the original input. The attack checker ensures that ARDILLA creates concrete exploits, which are much easier for programmers to fix than reports of abstract traces [9,37].

The attack checker is different for SQLI and XSS. In both types of Web attacks, the PHP program interacts with another component (a database or a Web browser) in a way the programmer did not intend. The essence of an SQLI attack is a change in the structure of the SQL statement that preserves its syntactic validity (otherwise, the database rejects the statement and the attack attempt is unsuccessful) [107]. The essence of an XSS attack is the introduction of additional script-inducing constructs (e.g., `<script>` tags) into a dynamically-generated HTML page [115].

ARDILLA detects attacks by comparing surely-innocuous runs with potentially-malicious ones. We assume that the input generator creates innocuous (non-attack) inputs, since the values of the input parameters are simple constants such as `1` or literals from the program text. Therefore, the innocuous input represents how the program is intended to interact with a component (database or browser). The attack generator creates potentially-malicious inputs.

The checker runs the program on the two inputs and compares the executions. Running the program on the attack-candidate input avoids two potential sources of false warnings: (i) input sanitizing—the program may sanitize (i.e., modify to make harmless) the input before passing it into a sensitive sink. ARDILLA does not require the user to specify a list of sanitizing routines. (ii) input filtering—the program may reject inputs that satisfy a malicious-input pattern (blacklisting), or else fail to satisfy an innocuous-input pattern (whitelisting). However, the symbolic expressions are unaffected by control flow (symbolic expressions only reflect data flow) and cannot capture input filtering.

The **SQLI attack checker** compares database statements (e.g., `SELECT`, `INSERT`) issued by the PHP program executed separately on the two inputs. The checker compares the first pair of corresponding statements, then the second, etc. The checker signals an attack if the statements in any pair are both valid SQL but have different syntactic structure (i.e., parse tree).

The **XSS attack checker** signals an attack if the HTML page produced from the execution of a candidate attack input (or sequence of inputs, for second-order attacks) contains additional script-inducing constructs.

5.3.5 Concolic Database

The concolic database stores both concrete and symbolic values for each data record. In a Web application, the database is a shared state that enables the exchange of data between users. The concolic database tracks the flow of user-provided data between *different runs* of the PHP program and it is critical in creating second-order XSS attacks.

The concolic database is implemented as a duplicate of the concrete database, with each table having additional columns that store symbolic data. ARDILLA uses these columns to store symbolic expressions.

msg	**topicid**	**msg_s**	**topicid_s**
Test message	1	∅	∅
Hello	2	msg	topicid

Figure 5-5: Example state of the concolic database table messages used by the PHP program of Figure 5-1. Each concrete column (left-most two columns) has a symbolic counterpart (right-most two columns) that contains a symbolic expression. The ∅ values represent empty symbolic expressions.

Figure 5-5 shows an example database state during the execution of the program in Figure 5-1. Assume the database was pre-populated with a test message in topic 1, so the symbolic expressions for fields in the first row are empty. When the user posts a message Hello in topic 2 (line 28), the symbolic expressions from the respective input parameters are stored along with their concrete values in the second row. Later, when the user fetches data from that row (line 43), the symbolic expressions are also fetched and propagated to the assigned variables.

ARDILLA dynamically rewrites SQL statements in the program under test. The rewritten SQL statements account for the new columns—either updating or reading symbolic expressions. Our current implementation handles a subset of SQL, rewriting their strings before passing them into mysql_query: CREATE TABLE, INSERT, UPDATE, and (non-nested) SELECT. (The DELETE statement and WHERE condition do not need to be rewritten—the database server can locate the relevant rows using the concrete values.)

- CREATE TABLE creates a new table. ARDILLA rewrites the statement to add a duplicate for each column (e.g., the two right-most columns in Figure 5-5) to use for storing symbolic expressions.
- INSERT adds new rows to tables. ARDILLA rewrites the statement to store symbolic expressions in the duplicate columns. For example, consider the following PHP string representing an SQL statement (PHP automatically performs the string concatenation):

```
INSERT INTO messages VALUES('$_GET['msg']',
 '$_GET['topicid']')
```

Consider an execution in which parameters msg and topicid have concrete values Hello and 2 and have symbolic expressions that contain only the parameters themselves. ARDILLA dynamically rewrites the statement as follows:

```
INSERT INTO messages VALUES('Hello','2',
 'msg','topicid')
```

- UPDATE modifies values in tables. For example, for:

```
UPDATE messages SET msg='$_GET['msg']'
 WHERE topicid='$_GET['topicid']'
```

ARDILLA's dynamic rewriting for UPDATE is similar to that for INSERT (the WHERE condition is unchanged):

```
UPDATE messages SET msg='Hi',
 msg_s='msg' WHERE topicid='3'
```

• SELECT finds and returns table cells. ARDILLA rewrites the statement to include the duplicate (symbolic) column names in the selection. Thereafter, ARDILLA uses the value retrieved from the duplicate column as the symbolic expression for the concrete value retrieved from the original column. For example, consider the concrete statement executed in line 39 of the program in Figure 5-1 (given the example state of the concolic database in Figure 5-5).

```
SELECT msg FROM messages WHERE topicid = '2'
```

ARDILLA rewrites the statement to:

```
SELECT msg, msg_s FROM messages WHERE topicid = '2'
```

The result of executing this rewritten statement on the table in Figure 5-5 is a 1-row table with concrete string Hello and associated symbolic expression msg, in columns msg and msg_s.

5.4 Evaluation

We evaluated ARDILLA on five open-source programs. We downloaded the programs from http://sourceforge.net: schoolmate 1.5.4 (tool for school administration, 8181 lines of code, or LOC), webchess 0.9.0 (online chess game, 4722 LOC), faqforge 1.3.2 (tool for creating and managing documents, 1712 LOC), EVE 1.0 (player-activity tracker for an online game, 915 LOC), and geccbblite 0.1 (a simple bulletin board, 326 LOC). We used the latest available versions as of 5 September 2008.

We performed the following procedure for each subject program.

1. Run the program's installation script to create the necessary database tables.

2. Pre-populate the database with representative data (e.g., defaults where available).

3. Run ARDILLA with a 30-minute time limit in each of three modes: SQLI, first-order XSS, and second-order XSS. The time limit includes all experimental tasks, i.e., input generation, concolic execution, attack generation (including string-constraint solving), and attack checking. When necessary, we provided the input generator with (non-administrator) username and password combinations. Doing so poses no methodological problems because an attacker can use a legitimate account to launch an attack. The SQLI mode used the HAMPI string-constraint solver, while the XSS modes used the attack-pattern library.

4. Manually examine attack vectors reported by ARDILLA to determine if they reveal true security vulnerabilities. We did not know any SQLI or XSS vulnerabilities in the subject programs before performing the experiments. (Thanks to previous studies [114,115], we were *aware of* the presence of first-order XSS and SQLI vulnerabilities in **geccbblite** and **EVE**.)

We ran ARDILLA in two modes for checking validity of XSS attacks: lenient and strict. (The SQLI checker has only one mode.) In the lenient mode, the XSS checker reports a vulnerability when the outputs differ in script-inducing elements or HTML elements such as `href`. In the strict mode, the XSS checker only reports a vulnerability when the outputs differ in script-inducing elements.

5.4.1 Measurements

Number of sensitive sinks (all) is the statically computed number of `echo`/`print` (for XSS) or `mysql_query` statements (for SQLI), whose parameter is *not a constant string*.
Number of reached sinks (reach) on all generated inputs is an indication of coverage achieved by the input generator. This measure is suitable for ARDILLA, because ARDILLA looks for attacks on sensitive sinks.
Number of tainted sinks (taint) is the number of sensitive sinks reached *with non-empty symbolic expressions* during concolic execution. Each such occurrence *potentially* exposes a vulnerability, which ARDILLA uses the attack generator and checker to test.
Number of validated vulnerabilities (Vuln): We count at most *one* vulnerability per sensitive sink, since a single-line code-fix would eliminate all attacks on the sink. If a single attack vector attacks multiple sensitive sinks, then we examine and count each vulnerability separately. This number does not include false positives. We manually inspected each ARDILLA report and determined whether it really constituted an attack (i.e., corruption or unintended disclosure of data for SQL, and unintended HTML structure for XSS). For second-order XSS, we checked that the attacker's malicious input can result in an unintended Web page for the victim.
Number of false positives (F) is the number of ARDILLA reports that are *not* validated vulnerabilities.
HAMPI performance: we measured solving time for each string constraint and each size of $N = 1 \dots 6$. (HAMPI found all known solutions for $N \leq 6$.)

5.4.2 Results

ARDILLA found 23 SQLI, 33 first-order XSS, and 12 second-order XSS vulnerabilities in the subject programs (see Figure 5-6). The attacks that ARDILLA found, as well as the attack patterns we used, are available at `http://pag.csail.mit.edu/ardilla`.

We examined two of the three instances in which ARDILLA found no vulnerabilities. In **geccbblite**, we manually determined that there are no first-order XSS

		sensitive sinks			lenient		strict	
program	mode	all	reach	taint	Vuln	F	Vuln	F
schoolmate	SQLI	218	28	23	**6**	0	**6**	0
	XSS1	122	26	20	**14**	6	**10**	0
	XSS2	122	4	4	**4**	0	**2**	0
webchess	SQLI	93	42	40	**12**	0	**12**	0
	XSS1	76	39	39	**13**	18	**13**	0
	XSS2	76	40	0	**0**	0	**0**	0
faqforge	SQLI	33	7	1	**1**	0	**1**	0
	XSS1	35	10	4	**4**	0	**4**	0
	XSS2	35	0	0	**0**	0	**0**	0
EVE	SQLI	12	6	6	**2**	0	**2**	0
	XSS1	24	5	4	**2**	0	**2**	0
	XSS2	24	5	3	**3**	0	**2**	0
geccbblite	SQLI	10	8	6	**2**	0	**2**	0
	XSS1	17	17	11	**0**	0	**0**	0
	XSS2	17	17	5	**5**	0	**4**	0
Total	SQLI	366	91	76	**23**	0	**23**	0
	XSS1	274	97	78	**33**	24	**29**	0
	XSS2	274	66	12	**12**	0	**8**	0

Figure 5-6: Results of running ARDILLA to create SQLI, XSS1 (first-order XSS), and XSS2 (second-order XSS) attacks. The lenient and strict columns refer to ARDILLA modes (Section 5.4). Section 5.4.1 describes the remaining columns (Vuln columns in bold list the discovered real vulnerabilities).

vulnerabilities. In **faqforge**, we manually determined that each database write requires administrator access, so there are no second-order XSS vulnerabilities. (We did not manually inspect **webchess** for second-order XSS attacks, due to the program's size and our unfamiliarity with the code.)

We examined all 23 SQLI reports issued by ARDILLA and found no false positives. All attacks involved disrupting the SQL WHERE clause. In 4 cases, attacks result in data corruption; in 19 cases, attacks result in information leaking, sometimes as serious as bypassing login authentication.

We examined all 69 (33+24+12) unique XSS reports issued by ARDILLA. We found 24 false positives in the lenient mode for first-order XSS (42% false-positive rate), and 0% percent false-positive rate for all other cases: strict first-order XSS, lenient and strict second-order XSS.

We examined cases of tainted sinks for which ARDILLA did not create attacks. The most common reason is that the same sink may not be reachable with a malicious input because of control-flow filtering that concolic execution does not capture. Such reachability is very hard to determine manually from program text. The existence of such those cases shows that not all tainted sinks may be exploitable and that ARDILLA's attack checker is useful in identifying exploitable vulnerabilities.

The HAMPI string-constraint solver was effective in creating attack candidates. ARDILLA generated 304 HAMPI constraints, each of which was built from the execution of a particular path through an application. Each HAMPI constraint was created from a combination of one of 4 attack patterns and one of 76 attack-candidate queries (one per tainted sink). HAMPI attempted solving the constraints for each tainted sink for increasing sizes of the string variable, until either a solution was found (for any attack pattern), or a timeout of 120 seconds per constraint was reached. The SQL grammar we used encoded each terminal using a single unique character, so the variable sizes for HAMPI constraints were not the same as the sizes of concrete attack values. ARDILLA created attack-candidate values by expanding the terminals in the computed solutions. ARDILLA checked each attack candidate using the SQLI attack checker. ARDILLA found the following number of attacks per variable size: 0 attacks for sizes 0...4, 14 attacks for sizes ≤ 5, 23 attacks for size 6. ARDILLA found no additional attacks for variable sizes greater than 6.

Example Created SQLI Attack

In **webchess**, ARDILLA found a vulnerability in `mainmenu.php` that allows an attacker to retrieve information about all players without entering a password. The application constructs the vulnerable statement directly from user input:

```
"SELECT * FROM players WHERE nick = '" . $_POST['txtNick']
. "' AND password = '" . $_POST['pwdPassword'] . "'"
```

The attack vector contains the following two crucial parameters (others omitted for brevity)

```
ToDo    → NewUser
txtNick → foo' or 1=1 --
```

which causes execution to construct the following malicious SQL statement which bypasses authentication (-- starts an SQL comment):

```
SELECT * FROM players WHERE nick = 'foo' or 1=1 --
' AND password = ''
```

Comparison to Previous Studies

Two of our subject programs were previously analyzed for vulnerabilities. In **geccbblite**, a previous study [115] found 1 first-order XSS vulnerability, and 7 second-order XSS vulnerabilities (possibly including false positives). However, ARDILLA and our manual examination of **geccbblite** found no first-order XSS vulnerabilities. In **EVE**, another study [114] found 4 SQLI vulnerabilities. The result data from neither study are available so we cannot directly compare the findings or establish the reason for the inconsistencies.

Comparison to Black-box Fuzzing

We compared ARDILLA to a black-box fuzzer Burp Intruder[5] (listed among the 10 most popular Web-vulnerability scanners[6]). Burp Intruder is a fuzzer for finding first-order XSS attacks. We configured the fuzzer according to its documentation. The fuzzer requires manual setting up of HTTP request patterns to send to the Web application. Additionally, the fuzzer requires manual indication of variables to mutate. This is hard because it requires examining the source to find names of parameters read from the `$_GET` and `$_POST` arrays. We ran the fuzzer using the same attack-pattern library that ARDILLA uses, and on the same subject programs. (We have not been able to successfully configure **webchess** to run with the fuzzer.) We ran the fuzzer until completion (up to 8 hours).

The fuzzer found 1 first-order XSS vulnerability in **schoolmate**, 3 in **faqforge**, 0 in **EVE**, and 0 in **geccbblite**. All 4 vulnerabilities reported by the fuzzer were also discovered by ARDILLA.

Limitations

The main limitation of ARDILLA stems from the input generator. ARDILLA can only generate attacks for a sensitive sink *if* the input generator creates an input that reaches the sink. However, effective input generation for PHP is challenging [2,77,116], complicated by the dynamic language features and execution model (running a PHP program often generates an HTML page with forms and links that require user interaction to execute code in additional files). In particular, the generator that ARDILLA uses can create inputs only for one PHP script at a time and cannot simulate sessions (i.e., user–application interactions that involve multiple pages), which is a serious hindrance to achieving high coverage in Web applications; line coverage averaged less than 50%. In fact, only on *one* application (**webchess**) did the input generator run until the full 30-minute time limit—in all other cases, the generator finished within 2 minutes because it did not manage to cover more code. We also attempted to run the generator on a larger application, the **phpBB** Web-forum creator (35 kLOC), but it achieved even lower coverage (14%). ARDILLA uses the input generator as a black box and any improvement in input generation (such as proposed recently by Artzi et al. [3]) is likely to improve ARDILLA's effectiveness.

5.5 Related Work

We describe previous approaches to securing Web applications from input-based attacks.

Defensive coding relies on special libraries to create safe SQL queries [25,78]. Defensive coding can, in principle, prevent all SQLI attacks. The technique is suit-

[5] `http://portswigger.net/intruder`

[6] `http://sectools.org/web-scanners.html`

able for new code. However, it requires rewriting existing code, while our technique requires no change to the programming language, the libraries, or the application.

Static approaches can, in principle, prove the *absence* of vulnerabilities [70, 113–115, 117]. In practice, however, analysis imprecision causes false warnings. Additionally, static techniques do not create concrete attack vectors. In contrast, our technique does not introduce such imprecision, and creates attack vectors.

Dynamic monitoring aims to prevent SQLI attacks by tracking user-provided values [52, 87, 92, 107] during operation of a deployed application. However, dynamic monitoring does not help to remove errors before software deployment, and requires either modifying the application, or running a modified server. For example, CANDID [6] modifies the application source and requires changing the runtime system, with performance overhead of up to 40% on the production application.

Information-flow control restricts the flow of information between pieces of software, either statically [98] or dynamically [119, 121]. Information-flow control enforces confidentiality and integrity policies on the data and prevents attacks that use inappropriate information flows. However, some SQLI and XSS attacks abuse legitimate information flows; the SQL queries or the JavaScript can be dynamically generated and can depend on legal user input. Information-flow control requires modifying the application and either the operating system and the libraries, or the programming language. System-level techniques may have runtime performance overhead up to 40% [119].

Static and dynamic approaches can be combined [51, 56]. Lam et al. [68] combine static analysis, model checking, and dynamic monitoring. QED [75] combines static analysis and model checking to automatically create SQLI and first-order XSS attacks on Java applications. In contrast to ARDILLA, QED (i) does not target second-order XSS, and (ii) requires programmers to use a custom specification language to describe attacks.

Saner [5] combines static and dynamic analyses to find potential XSS and SQLI vulnerabilities. Saner focuses on the sanitization process and abstracts away other details of the application, i.e., Saner creates attack vectors only for extracted, possibly infeasible, paths from the static dependency graph (Saner does dynamically validate the exploitability of string-manipulating code from those paths, but ignores control flow). Saner also reports a vulnerability whenever a path from source to sink contains no custom sanitation. The path, however, may be infeasible or not exploitable. Saner tests each source-to-sink path independently and may miss attacks in which output is constructed from multiple sinks. To detect attacks, Saner simply searches for specific strings in the output, whereas ARDILLA compares the structure of HTML or SQL between innocuous and attack runs.

Apollo [2] generates test inputs for PHP, checks the execution for crashes, and validates the output's conformance to HTML standards. The goal of ARDILLA is different: to find security vulnerabilities. ARDILLA uses the test-input generator subcomponent of Apollo as a black box. ARDILLA's taint propagation implementation is partially based on that of Apollo, but we enhanced it significantly by adding propagation across function calls, taint filters, taint sinks, and tracing taint across database calls.

Emmi et al. [36] model a database using symbolic constraints and provide a custom string-constraint solver to create database states that help exercise various execution paths in the Web application. Our work differs in its goal (finding security vulnerabilities vs. improving test coverage) and in the targeted language (PHP vs. Java).

Wassermann et al.'s tool [116] executes a PHP application on a concrete input and collects symbolic constraints. Upon reaching an SQL statement, the tool attempts to create an input that exposes an SQL injection vulnerability, by using a string analysis [83]. The tool has re-discovered 3 previously known vulnerabilities. The most important differences between Wassermann's work and ours are: (i) Their tool has not discovered any previously unknown vulnerabilities, and requires a precise indication of an attack point. Our tool has discovered 68 previously unknown vulnerabilities and requires no indication of vulnerable points. (ii) Their technique focuses on SQLI, while ours targets both SQLI and XSS. (iii) Their tool performs source-code instrumentation and backward-slice computation by re-executing and instrumenting additional code. Our tool works on unchanged application code. (iv) Their tool requires manual loading of pages and supplying of inputs to the page, while ours is fully automatic.

5.6 Conclusion

We have presented concolic security testing, a technique for creating SQL injection and cross-site scripting (XSS) attacks in Web applications and an automated tool, ARDILLA, that implements the technique for PHP. Our technique is based on input generation, concolic execution, and input mutation. To find a variant of the input that exposes a vulnerability, input mutation uses the HAMPI string-constraint solver, or a library of attack patterns. Using a novel concolic database to store symbolic expressions, ARDILLA can effectively and accurately find the most damaging type of input-based Web application attack: stored (second-order) XSS. A novel attack checker that compares the output from running on an innocuous input and on a candidate attack vector allows ARDILLA to detect vulnerabilities with high accuracy. In our experiments, ARDILLA found 68 attack vectors in five programs, each exposing a different vulnerability, with few false positives.

Chapter 6

Conclusions

This chapter presents the summary of the contributions of this dissertation, discusses lessons learned (Section 6.1), and proposes directions for future work (Section 6.2).

Software systems are becoming steadily more complex and our reliance on them also increases. Testing is currently the dominant way of ensuring software reliability and we think that it is likely to remain so. Regardless of its well-known deficiencies, e.g., labor-intensity and incompleteness, testing is the easiest to adopt strategy for software reliability, and it can be very effective.

The cost of testing can be reduced by automation. Testing is expensive because creating effective test suites requires skilled test engineers. The aim of the research presented in this dissertation is to automate the process of creating effective test inputs.

Concolic testing is a paradigm of implementation-based software testing. Concolic testing is based on the premise that the software implementation can be exploited to find reliability errors. A number of automated concolic testing tools have been developed and shown to efficiently find errors in real software. This dissertation presents techniques and tools in the concolic testing paradigm.

The **key idea** of this dissertation is that by enhancing the underlying constraint solver with the theory of strings, concolic testing can be made much more effective. We presented such a string-constraint solver, and evaluate its efficiency. Furthermore, we showed the effectiveness of our idea by improving concolic testing of programs that have structured inputs (e.g., programs whose inputs come from a context-free grammar), and of Web applications that manipulate string inputs.

6.1 Discussion

Software testing has two main driving forces: quality and security. We feel that software quality is today higher than ever. Thanks partly to adoption of good testing practices (e.g., automated regression tests) and software-engineering practices (e.g., automated build systems) in the industry, the ease-of-use, feature repertoire, and general reliability of today's commonly-used software is impressive, despite the vast and very rapidly increasing amount of software we use every day. Software security, however, is a massive and growing problem. Thousands of severe vulnerabilities are reported every year, and their numbers are quickly increasing[1].

There is a transfer of ideas and technologies from research to hackers. Testing and analysis technologies that used to exist as research prototypes are now widely used by villains. For example, fuzz testing, developed in 1990 at the University of Wisconsin [82], has leapt to prominence in the 2000s as the main technology used by blackhats (as well as whitehat security professionals[2]). The transfer of more sophisticated technologies, such as concolic testing, is very likely and perhaps imminent[3].

The arms race between researchers and hackers will lead to more automated technologies and more effective tools. We think that progress in this area will be fueled by combining implementation-based and specification-based testing, by powerful constraint solvers that combine many theories, and by contests of error-finding tools.

Exploiting the knowledge of input-format specifications and combining it with implementation-based analyses (Concolic Security Testing is an example of such a combination), will play an important role in future testing tools. This combined approach compensates the inherent limitations of each analyses with the strengths of the other analyses. Effective testing tools will require powerful constraint solvers that combine many theories: strings, integers, bit vectors, arrays, functions, etc. Creating software error-finding contests (such as the Iron Chef contest[4] or the SAMATE project[5]) will be important in fostering development of sophisticated tools. Similar competitions have proved to be very valuable in many fields of computer science, from formal methods[6, 7] to bioinformatics[8].

We feel that the falsification view (i.e., finding errors) will continue and expand its dominance over the validation view (i.e., proving the absence of errors) of software reliability. We think that the error-finding approach is likely to become important even in domains, such as security, that traditionally have been viewed mostly with validation in mind. Validation techniques, such as static analysis, will be important as preludes or guides for test-input generation [28]. Similarly, bounded verification [34, 59] is likely to grow in significance, provided it is combined with generation of concrete inputs that demonstrate real errors. We think that automatic software verification is desirable and it may become feasible, but it is going to be achievable only for small, critical, pieces of software architecture, e.g., common data structures [120] or the Java bytecode verifier.

[1] `http://nvd.nist.gov`

[2] `http://browserfun.blogspot.com`

[3] `http://www.unprotectedhex.com/psv`

[4] `http://www.blackhat.com`

[5] `http://samate.nist.gov`

[6] `http://www.satcompetition.org`

[7] `http://www.smtcomp.org`

[8] `http://predictioncenter.org`

6.2 Future Work

We present several ideas for future work on research presented in this dissertation.

Richer grammars in HAMPI. Our results have shown that using context-free grammars is effective in testing programs with structured inputs, and in security-testing of Web applications. Certain applications, however, require more expressive grammars. For example, type-correct Java programs do not form a context-free language. Thus, to effectively use concolic testing on a Java compiler (i.e., test the compiler), a more expressive string-constraint solver is required, or else grammar-based concolic testing generates mostly type-incorrect programs. HAMPI always fixed-sizes the size of string variables which, strictly speaking, limits HAMPI to regular languages. Therefore, fixed-sizing enables increasing the expressiveness of the solver without the loss of decidability. In particular, it would be interesting to extend HAMPI to context-sensitive languages [105] or to attribute grammars [88].

Richer string operations in HAMPI. In HAMPI, we implemented the most common string operations: concatenation, containment, and equality. Some programs, however, may use a repertoire of string operations that is richer, e.g., substring extraction, substring replacement, lexicographic comparison, length checking. Programs use such operations, for example, in input sanitization. Therefore, it is important for a string-constraint solver to support those operations [12,41,61].

Integration of HAMPI with other theories. HAMPI handles only string constraints. Programs, however, manipulate string inputs as well as integers, floating-point numbers, etc. Integration of string constraints into an SMT solver would enable handling multiple kinds of input values uniformly [12,41]. An alternative is to enhance HAMPI itself with other theories, e.g., linear arithmetic.

Unbounded variable length in HAMPI. String variables in HAMPI are of fixed lengths, which allows the solver to handle context-free grammars, and enables efficient encoding in bit-vector logic. HAMPI could be enhanced to detect, in certain cases, that no solution, of any length, exists. For example, if a HAMPI input does not contain a context-free grammar, then unbounded (un)satisfiability can be established. This enhancement would make HAMPI even more applicable to program analysis and verification.

Multiple string variables in HAMPI. Currently, HAMPI allows only a single string variable. Handling multiple string variables would enhance testing of programs that handle multiple inputs. For example, PHP programs have multiple inputs (each key-value pair can be seen as an input), and HAMPI solves for each key in turn. Handling multiple fixed-size variables is straightforward but if the variables are of bounded lengths, then the solver needs to account for the exponential number of all possible variable-size combinations.

State-space exploration in ARDILLA. The fundamental limitation of our current concolic security testing tool for PHP stems from the input generator. The current input generator always starts in the same state of the application (i.e., same state of the database). Thus, the generator simulates only those user inputs that involve a single interaction. However, Web applications operate in sessions, i.e., series of user interactions in which a PHP script generates an HTML page which contains a Web form, which calls another PHP script, etc. An efficient input generator that handles multi-step user interactions [3] could increase effectiveness of ARDILLA.

Non-pattern-based attack definitions in ARDILLA. Currently, ARDILLA requires specifying SQLI attacks as patterns, e.g., "the generated SQL query should not contain the `'1'='1'` tautology in the `WHERE` clause". This approach is effective but limited by the pattern library. A potential, more expressive approach would be to define an attack as a query in which the user-controlled part of the input is not contained in a sub-tree of the parse [107].

Bibliography

[1] César L. Alonso, David Alonso, Mar Callau, and José Luis Montaña. A word equation solver based on Levensthein distance. In *Mexican International Conference on Artificial Intelligence*, 2007.

[2] Shay Artzi, Adam Kieżun, Julian Dolby, Frank Tip, Danny Dig, Amit Paradkar, and Michael Ernst. Finding bugs in dynamic Web applications. In *International Symposium on Software Testing and Analysis*, 2008.

[3] Shay Artzi, Adam Kieżun, Julian Dolby, Frank Tip, Danny Dig, Amit Paradkar, and Michael D. Ernst. Finding bugs in web applications using dynamic test generation and explicit state model checking. Technical Report MIT-CSAIL-TR-2009-010, Massachusetts Institute of Technology Computer Science and Artificial Intelligence Laboratory, 2009.

[4] Roland Axelsson, Keijo Heljank, and Martin Lange. Analyzing context-free grammars using an incremental SAT solver. In *International Colloquium on Automata, Languages and Programming*, 2008.

[5] Davide Balzarotti, Marco Cova, Victoria Felmetsger, Nenad Jovanovic, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. Saner: Composing static and dynamic analysis to validate sanitization in Web applications. In *IEEE Symposium on Security and Privacy*, 2008.

[6] Sruthi Bandhakavi, Prithvi Bisht, P. Madhusudan, and V. N. Venkatakrishnan. CANDID: preventing SQL injection attacks using dynamic candidate evaluations. In *Conference on Computer and Communications Security*, 2007.

[7] Adam Barth, Juan Caballero, and Dawn Song. Secure content sniffing for web browsers or how to stop papers from reviewing themselves. In *IEEE Symposium on Security & Privacy*, 2009.

[8] Marco Benedetti. sKizzo: a Suite to Evaluate and Certify QBFs. In *Conference on Automated Deduction*, 2005.

[9] Nicolas Bettenburg, Sascha Just, Adrian Schröter, Cathrin Weiss, Rahul Premraj, and Thomas Zimmermann. What makes a good bug report? In *International Symposium on the Foundations of Software Engineering*, 2008.

[10] Armin Biere. Resolve and expand. In *International Conference on Theory and Applications of Satisfiability Testing*, 2005.

[11] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, Ofer Strichman, and Yunshan Zhu. Bounded model checking. *Advances in Computers*, 58, 2003.

[12] Nikolaj Bjørner, Nikolai Tillmann, and Andrei Voronkov. Path feasibility analysis for string-manipulating programs. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2009.

[13] Nikita Borisov, David Brumley, Helen J. Wang, and Chuanxiong Guo. Generic application-level protocol analyzer and its language. In *Network and Distributed System Security Symposium*, 2007.

[14] Chandrasekhar Boyapati, Sarfraz Khurshid, and Darko Marinov. Korat: automated testing based on Java predicates. *International Symposium on Software Testing and Analysis*, 2002.

[15] Juan Caballero, Stephen McCamant, Adam Barth, and Dawn Song. Extracting models of security-sensitive operations using string-enhanced white-box exploration on binaries. Technical Report UCB/EECS-2009-36, EECS Department, University of California, Berkeley, 2009.

[16] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Symposium on Operating Systems Design and Implementation*, 2008.

[17] Cristian Cadar and Dawson R. Engler. Execution generated test cases: How to make systems code crash itself. In *SPIN Workshop on Model Checking of Software*, 2005.

[18] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. EXE: automatically generating inputs of death. In *Conference on Computer and Communications Security*, 2006.

[19] Cenzic. Application security trends report Q1 2008. `http://www.cenzic.com`.

[20] Aske Simon Christensen, Anders Møller, and Michael I. Schwartzbach. Precise analysis of string expressions. In *International Static Analysis Symposium*, 2003.

[21] Koen Claessen and John Hughes. QuickCheck: A lightweight tool for random testing of Haskell programs. In *International Conference on Functional Programming*, 2000.

[22] Edmund M. Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ANSI-C programs. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2004.

[23] Edmund M. Clarke, Daniel Kroening, Natasha Sharygina, and Karen Yorav. Predicate abstraction of ANSI-C programs using SAT. *Formal Methods in System Design*, 25(2-3), 2004.

[24] Byron Cook, Daniel Kroening, and Natasha Sharygina. Cogent: Accurate theorem proving for program verification. In *International Conference on Computer Aided Verification*, 2005.

[25] William R. Cook and Siddhartha Rai. Safe query objects: statically typed objects as remotely executable queries. In *International Conference on Software Engineering*, 2005.

[26] David Coppit and Jiexin Lian. yagg: an easy-to-use generator for structured test inputs. *Automated Software Engineering*, 2005.

[27] CREST: automatic test generation tool for C. `http://code.google.com/p/crest`.

[28] Christoph Csallner and Yannis Smaragdakis. DSD-Crasher: A hybrid analysis tool for bug finding. *ISSTA*, 2007.

[29] Weidong Cui, Jayanthkumar Kannan, and Helen J. Wang. Discoverer: Automatic protocol reverse engineering from network traces. In *USENIX Security Symposium*, 2007.

[30] Brett Daniel, Danny Dig, Kely Garcia, and Darko Marinov. Automated testing of refactoring engines. In *International Symposium on the Foundations of Software Engineering*, 2007.

[31] Robert Dąbrowski and Wojciech Plandowski. On word equations in one variable. In *International Symposium on Mathematical Foundations of Computer Science*, 2002.

[32] Robert Dąbrowski and Wojciech Plandowski. Solving two-variable word equations. *Lecture Notes in Computer Science*, 2004.

[33] Leonardo de Moura and Nikolaj Bjørner. Z3: An Efficient SMT Solver. In *International Conference on Tools and Algorithms for the construction and Analysis of Systems*, 2008.

[34] Greg Dennis, Kuat Yessenov, and Daniel Jackson. Bounded verification of voting software. *Proceedings of the Second International Conference on Verified Software: Theories, Tools, Experiments*, 2008.

[35] Niklas Eén and Niklas Sörensson. An extensible SAT solver. In *International Conference on Theory and Applications of Satisfiability Testing*, 2003.

[36] Michael Emmi, Rupak Majumdar, and Koushik Sen. Dynamic test input generation for database applications. In *International Symposium on Software Testing and Analysis*, 2007.

[37] Dawson R. Engler and Madanlal Musuvathi. Static analysis versus software model checking for bug finding. In *International Conference on Verification, Model Checking, and Abstract Interpretation*, 2004.

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

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

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

[41] Xiang Fu, Xin Lu, Boris Peltsverger, Shijun Chen, Kai Qian, and Lixin Tao. A static analysis framework for detecting SQL injection vulnerabilities. In *International Computer Software and Applications Conference*, 2007.

[42] Vijay Ganesh and David L. Dill. A decision procedure for bit-vectors and arrays. In *International Conference on Computer Aided Verification*, 2007.

[43] Patrice Godefroid. The soundness of bugs is what matters (position statement). In *Workshop on the Evaluation of Software Defect Detection Tools*, 2005.

[44] Patrice Godefroid. Compositional Dynamic Test Generation. In *Symposium on Principles of Programming Languages*, 2007.

[45] Patrice Godefroid, Adam Kieżun, and Michael Y. Levin. Grammar-based whitebox fuzzing. In *Programming Language Design and Implementation*, 2008.

[46] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: Directed automated random testing. In *Programming Language Design and Implementation*, 2005.

[47] Patrice Godefroid, Michael Y. Levin, and David A. Molnar. Active property checking. In *International Conference on Embedded Systems*, 2008.

[48] Patrice Godefroid, Michael Y. Levin, and David A. Molnar. Automated whitebox fuzz testing. In *Network and Distributed System Security Symposium*, 2008.

[49] Keith Golden and Wanlin Pang. Constraint reasoning over strings. In *Constraint Programming*, 2003.

[50] Sumit Gulwani, Saurabh Srivastava, and Ramarathnam Venkatesan. Program analysis as constraint solving. In *Programming Language Design and Implementation*, 2008.

[51] William G. J. Halfond and Alessandro Orso. AMNESIA: Analysis and Monitoring for NEutralizing SQL-Injection Attacks. In *Automated Software Engineering*, 2005.

[52] William G. J. Halfond, Alessandro Orso, and Pete Manolios. WASP: Protecting Web applications using positive tainting and syntax-aware evaluation. *IEEE Transactions on Software Engineering*, 34(1), 2008.

[53] Kenneth V. Hanford. Automatic generation of test cases. *IBM Systems Journal*, 9(4), 1970.

[54] Pieter Hooimeijer and Westley Weimer. A decision procedure for subset constraints over regular languages. In *Programming Language Design and Implementation*, 2009.

[55] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to automata theory, languages and computation*. Addison-Wesley Series in Computer Science, 1979.

[56] Yao-Wen Huang, Fang Yu, Christian Hang, Chung-Hung Tsai, Der-Tsai Lee, and Sy-Yen Kuo. Securing Web application code by static analysis and runtime protection. In *International World Wide Web Conference*, 2004.

[57] Research Triangle Institute. The economic impacts of inadequate infrastructure for software testing. *National Institute of Standards and Technology RTI Project*, 2002.

[58] Daniel Jackson. Automating first-order relational logic. In *International Symposium on Foundations of Software Engineering*, 2000.

[59] Daniel Jackson and Mandana Vaziri. Finding bugs with a constraint solver. In *International Symposium on Software Testing and Analysis*, 2000.

[60] Karthick Jayaraman, David Harvison, Vijay Ganesh, and Adam Kieżun. jFuzz: A concolic whitebox fuzzer for Java. In *NASA Formal Methods Symposium*, 2009.

[61] Susmit Kumar Jha, Sanjit A. Seshia, and Rhishikesh Shrikant Limaye. On the computational complexity of satisfiability solving for string theories. Technical Report UCB/EECS-2009-41, EECS Department, University of California, Berkeley, 2009.

[62] Sarfraz Khurshid and Darko Marinov. TestEra: Specification-based testing of Java programs using SAT. *Automated Software Engineering*, 11(4), 2004.

[63] Adam Kieżun, Vijay Ganesh, Philip J. Guo, Pieter Hooimeijer, and Michael D. Ernst. Hampi: A solver for string constraints. In *International Symposium on Software Testing and Analysis*, 2009.

[64] Adam Kieżun, Philip J. Guo, Karthik Jayaraman, and Michael D. Ernst. Automatic creation of SQL injection and cross-site scripting attacks. In *International Conference on Software Engineering*, 2009.

[65] James C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7), 1976.

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

[67] Nils Klarlund and Michael I. Schwartzbach. A domain-specific language for regular sets of strings and trees. *IEEE Transactions on Software Engineering*, 25(3), 1999.

[68] Monica S. Lam, Michael Martin, Benjamin Livshits, and John Whaley. Securing Web applications with static and dynamic information flow tracking. In *Symposium on Partial Evaluation and Semantics-based Program Manipulation*, 2008.

[69] Ralf Lämmel and Wolfram Schulte. Controllable combinatorial coverage in grammar-based testing. *TestCom*, 2006.

[70] Benjamin Livshits and Monica Lam. Finding security vulnerabilities in Java applications with static analysis. In *USENIX Security Symposium*, 2005.

[71] Rupak Majumdar and Koushik Sen. LATEST: Lazy dynamic test input generation. Technical Report UCB/EECS-2007-36, EECS Department, University of California, Berkeley, 2007.

[72] Rupak Majumdar and Ru-Gang Xu. Directed test generation using symbolic grammars. In *Automated Software Engineering*, 2007.

[73] GS Makanin. The problem of solvability of equations in a free semigroup. *Sbornik: Mathematics*, 32(2), 1977.

[74] Brian A. Malloy and James F. Power. An interpretation of Purdom's algorithm for automatic generation of test cases. *International Conference on Computer and Information Science*, 2001.

[75] Michael Martin and Monica Lam. Automatic generation of XSS and SQL injection attacks with goal-directed model checking. In *USENIX Security Symposium*, 2008.

[76] Peter M. Maurer. Generating test data with enhanced context-free grammars. *IEEE Software*, 7(4), 1990.

[77] Sean McAllister, Engin Kirda, and Christopher Krügel. Leveraging user interactions for in-depth testing of Web applications. In *International Symposium on Recent Advances in Intrusion Detection*, 2008.

[78] Russell A. McClure and Ingolf H. Krüger. SQL DOM: compile time checking of dynamic SQL statements. In *International Conference on Software Engineering*, 2005.

[79] Bruce McKenzie. Generating strings at random from a context free grammar. Technical Report TR-COSC 10/97, Department of Computer Science, University of Canterbury, 1997.

[80] David Melski and Thomas Reps. Interconvertbility of set constraints and context-free language reachability. In *Symposium on Partial Evaluation and Semantics-based Program Manipulation*, 1997.

[81] Barton P. Miller, Lars Fredriksen, and Bryan So. An empirical study of the reliability of UNIX utilities. *Communications of the ACM*, 33(12), 1990.

[82] Barton P. Miller, Lars Fredriksen, and Bryan So. An empirical study of the reliability of UNIX utilities. *Communications of the ACM*, 33(12), 1990.

[83] Yasuhiko Minamide. Static approximation of dynamically generated Web pages. In *International World Wide Web Conference*, 2005.

[84] David A. Molnar and David Wagner. Catchconv: Symbolic execution and run-time type inference for integer conversion errors. Technical Report UCB/EECS-2007-23, EECS Department, University of California, Berkeley, Feb 2007.

[85] Robert C. Moore. Removing left recursion from context-free grammars. In *Proceedings of the first conference on North American chapter of the Association for Computational Linguistics*, 2000.

[86] Matthew W. Moskewicz, Concor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: engineering an efficient SAT solver. In *Design Automation Conference*, 2001.

[87] James Newsome and Dawn Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Network and Distributed System Security Symposium*, 2005.

[88] Jukka Paakki. Attribute grammar paradigms—a high-level methodology in language implementation. *ACM Computing Surveys*, 27(2), 1995.

[89] Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. Feedback-directed random test generation. In *International Conference on Software Engineering*, 2007.

[90] Ruoming Pang, Vern Paxson, Robin Sommer, and Larry Peterson. binpac: a yacc for writing application protocol parsers. *Proceedings of the 6th ACM SIGCOMM on Internet measurement*, 2006.

[91] Gilles Pesant. A regular language membership constraint for finite sequences of variables. In *International Conference on Principles and Practice of Constraint Programming*, 2004.

[92] Tadeusz Pietraszek and Chris Vanden Berghe. Defending against injection attacks through context-sensitive string evaluation. In *International Symposium on Recent Advances in Intrusion Detection*, 2005.

[93] Wojciech Plandowski. An efficient algorithm for solving word equations. In *Symposium on Theory of Computing*, 2006.

[94] Paul Purdom. A sentence generator for testing parsers. *BIT Numerical Mathematics*, 12(3), 1972.

[95] Claude-Guy Quimper and Toby Walsh. Global grammar constraints. In *International Conference on Principles and Practice of Constraint Programming*, 2006.

[96] Arcot Rajasekar. Applications in constraint logic programming with strings. In *Principles and Practice of Constraint Programming*, 1994.

[97] Hui Ruan, Jian Zhang, and Jun Yan. Test data generation for C programs with string-handling functions. In *Theoretical Aspects of Software Engineering Conference*, 2008.

[98] Andrei Sabelfeld and Andrew C. Myers. Language-based information-flow security. *Selected Areas in Communications*, 2003.

[99] Daniel J. Salomon and Gordon V. Cormack. Scannerless NSLR(1) parsing of programming languages. In *Programming Language Design and Implementation*, 1989.

[100] Meinolf Sellmann. The theory of grammar constraints. In *International Conference on Principles and Practice of Constraint Programming*, 2006.

[101] Koushik Sen and Gul Agha. CUTE and jCUTE: Concolic unit testing and explicit path model-checking tools. *Lecture Notes in Computer Science*, 4144, 2006.

[102] Koushik Sen, Darko Marinov, and Gul Agha. CUTE: A concolic unit testing engine for C. In *International Symposium on the Foundations of Software Engineering*, 2005.

[103] Daryl Shannon, Sukant Hajra, Alison Lee, Daiqian Zhan, and Sarfraz Khurshid. Abstracting symbolic execution with string analysis. In *Testing: Academic and Industrial Conference Practice and Research Techniques*, 2007.

[104] Ilya Shlyakhter, Manu Sridharan, Robert Seater, and Daniel Jackson. Exploiting subformula sharing in automatic analysis of qualified formulas. In *International Conference on Theory and Applications of Satisfiability Testing*, 2003.

[105] Michael Sipser. *Introduction to the Theory of Computation*. Course Technology, 1996.

[106] Emin Gün Sirer and Brian N. Bershad. Using production grammars in software testing. *Proceedings of the 2nd conference on Domain-specific languages*, 1999.

[107] Zhendong Su and Gary Wassermann. The essence of command injection attacks in Web applications. In *Symposium on Principles of Programming Languages*, 2006.

[108] Kevin Sullivan, Jinlin Yang, David Coppit, Sarfraz Khurshid, and Daniel Jackson. Software assurance by bounded exhaustive testing. In *International Symposium on Software Testing and Analysis*, 2004.

[109] Michael Sutton, Adam Greene, and Pedram Amini. *Fuzzing: Brute Force Vulnerability Discovery*. Addison-Wesley, 2007.

[110] Nikolai Tillmann and Jonathan de Halleux. Pex–White Box Test Generation for .NET. *Lecture Notes in Computer Science*, 4966, 2008.

[111] Emina Torlak and Daniel Jackson. Kodkod: A relational model finder. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2007.

[112] Mark Utting, Alexander Pretschner, and Bruno Legeard. A taxonomy of model-based testing. Technical Report 04/2006, Department of Computer Science, The University of Waikato, New Zealand, 2006.

[113] Gary Wassermann, Carl Gould, Zhendong Su, and Premkumar Devanbu. Static checking of dynamically generated queries in database applications. In *International Conference on Software Engineering*, 2004.

[114] Gary Wassermann and Zhendong Su. Sound and precise analysis of Web applications for injection vulnerabilities. In *Programming Language Design and Implementation*, 2007.

[115] Gary Wassermann and Zhendong Su. Static detection of cross-site scripting vulnerabilities. In *International Conference on Software Engineering*, 2008.

[116] Gary Wassermann, Dachuan Yu, Ajay Chander, Dinakar Dhurjati, Hiroshi Inamura, and Zhendong Su. Dynamic test input generation for web applications. In *International Symposium on Software Testing and Analysis*, 2008.

[117] Yichen Xie and Alex Aiken. Static detection of security vulnerabilities in scripting languages. In *USENIX Security Symposium*, 2006.

[118] Yichen Xie and Alex Aiken. Saturn: A scalable framework for error detection using Boolean satisfiability. In *International Conference on Computer Aided Verification*, 2007.

[119] Maxwell Krohn Alexander Yip, Micah Brodsky, Natan Cliffer, Frans Kaashoek, Eddie Kohler, and Robert Morris. Information flow control for standard OS abstractions. In *Symposium on Operating Systems Principles*, 2007.

[120] Karen Zee, Viktor Kuncak, and Martin C. Rinard. Full functional verication of linked data structures. In *Programming Language Design and Implementation*, 2008.

[121] Nickolai Zeldovich, Silas Boyd-Wickizer, and David Mazières. Securing distributed systems with information flow control. In *Symposium on Networked Systems Design and Implementation*, 2008.