Harvard University

Undergraduate Thesis

Security Analysis of Java Web Applications Using String Constraint Analysis

Author:
Louis Li

Supervisor:
Professor Stephen Chong

A thesis submitted in fulfillment of the requirements for the degree of Bachelor of Arts

in

Computer Science and Mathematics

April 2015

Contents

Abstract

Web applications are exposed to myriad security vulnerabilities related to malicious user string input. In order to detect such vulnerabilities in Java web applications, this project employs string constraint analysis, which approximates the values that a string variable in a program can take on. In string constraint analysis, program analysis generates string constraints – assertions about the relationships between string variables. We design and implement a dataflow analysis for Java programs that generates string constraints and passes those constraints to the CVC4 SMT solver to find a satisfying assignment of string variables. Using example programs, we illustrate the feasibility of the system in detecting certain types of web application vulnerabilities, such as SQL injection and cross-site scripting.

Acknowledgements

I would like to thank my three thesis readers, each having played a crucial role in my undergraduate experience as a computer scientist.

This work would not have been possible without my advisor, Professor Steve Chong. I had always been interested in programming languages since stumbling upon the seemingly esoteric programming language blog, Lambda the Ultimate. Through Steve's courses and advising, I was able to pursue my curiosity to the fullest. I appreciate the extraordinary amount of attention that he gives to undergraduate researchers and his initiative in uniting undergraduate thesis writers in computer science.

I would have been much worse prepared for the undertaking of a thesis without Professor Krszystof Gajos, who patiently guided me through my first major research project. He has provided me with invaluable advice for both maturing as a researcher – guiding me in the right direction, but always encouraging autonomy – and as an individual – pushing me to find my "superpowers" outside of academia.

Finally, I am grateful to Professor Greg Morrisett, whose undergraduate compilers course provided a crucial foundation for my understanding of dataflow analysis. Many of your students, including me, pick up your passion for the beauty of functional programming.

This project was made possible by the incredible patience of Andrew Johnson. It depended on his work with the Accrue Bytecode analysis framework. More importantly, I am thankful of the time that he took to answer my flood of questioning emails.

I cherish the opportunities provided by the Harvard Computer Science department. The faculty is incredibly supportive, and I believe that a positive undergraduate research experience is within grasp for any student in the department.

Thank you to Ruth Fong for her encouragement and support in my quest to become a computer scientist.

I owe my life to my family, who is forever supportive of my endeavors – especially research. Thank you, Mom, Dad, and Richard.

Chapter 1

Introduction

Web applications are exposed to myriad security vulnerabilities related to malicious user string input. Web servers often accept arbitrary user input through a variety of sources, such as form fields, URL parameters, and cookies. Common examples of such vulnerabilities are SQL injection, where an attacker can manipulate the database, or cross-site scripting, where an attacker can execute arbitrary code in a user's browser.

These security flaws can potentially be prevented by analyzing the code of the web application beforehand. In order to detect such vulnerabilities in Java web applications, this project employs *string constraint analysis*, which approximates the values that a string variable in a program can take on. A string constraint asserts a relationship between string variables. For example, consider the following constraint for string-typed variables x, y, z. If we have the constraint that x is the concatenation of y, z, then we assert that the string values that x could take on is equivalent to the set of string values that y could take on concatenated with the set of string values that z could take on.

If we know what values a string variable can take on at a given point in time, then we may be able to detect a vulnerability. For example, consider a basic application that takes user input and submits it to the SQL server. If our analysis finds that the query string variable fits the form of a valid SQL query that allows the user to arbitrarily manipulate the database, then we conclude that the program contains a SQL injection vulnerability.

We design and implement a dataflow analysis for Java programs that generates string constraints using dataflow analysis. It translates these constraints to the language of a satisfiable modulo theory solver to find a satisfying assignment of the string variables in the program. Using example programs, we illustrate the feasibility of the system in detecting certain types of web application vulnerabilities.

1.1 Motivation

Web applications that store or manipulate user input are at risk of security vulnerabilities. This is extremely common. Many vulnerabilities caused by untrusted data, such as cross-site scripting and SQL injection attacks, are ranked in the top 10 most common web attacks by the Open Web Application Security Project [10].

Ultimately, these security vulnerabilities are caused by developer error in the design of the application. Sensitive strings may be improperly sanitized, allowing users to provide malicious strings that exploit some aspect of the application. Static analysis aims to analyze an application codebase without actually executing code, which can be a convenient way for developers to secure code. This project offers another approach to static analysis, leveraging the capabilities of satisfiable modulo theory solvers to detect potential vulnerabilities.

Chapter 2

Related work

2.1 String analysis

Past work has used string constraint analysis to analyze string expressions of programs and detect security vulnerabilities.

Christensen et al. design an algorithm that associates each string expression with a context-free grammar. The context-free grammar represents the set of strings that can be generated by a string expression [2]. In this case, the constraints are used to represent context-free grammars, which are eventually approximated with regular languages.

The result of this work was released as the Java String Analyzer (JSA), which computes the resulting automata for each string expression in a Java program. AMNESIA, a system for detecting SQL injection attacks in Java web applications, combines JSA string solving with runtime monitoring [5].

Fu et al. describe a formalism for string constraints called Simple Linear String Equations. They implement an algorithm to solve these constraints for Java. This system is packaged as the constraint solver SUSHI. They apply the constraint solver to XSS detection [3].

BEK uses a representation of symbolic finite automata with SMT solvers to develop a language and system for analyzing string sanitization functions, which are often the source of cross-site scripting vulnerabilities [6].

2.2 Static analysis of web applications

Much work has been done in security analysis of web applications using various static analysis techniques. Due to the sheer volume of work done in this area, the projects are not enumerated here.

A particularly relevant project is Framework For Frameworks (F4F). It uses taint analysis – tracking the flow of potentially sensitive information in a program – to support modern Java web application frameworks, such as Java EE and Struts. Similar to this project, it uses parts of the Watson Libraries for Analysis (WALA) framework [15]. A core part of the F4F project is creating an end-to-end system for static analysis of a web application, handling analysis of difficult portions of frameworks such as XML configurations.

2.3 String solvers

Although this work does not directly explore techniques for SMT solvers, theorem proving, and string formulas, it leverages existing work in string solvers. This work uses the string theory capabilities of the SMT solver CVC4 [1]. Support for the string theories was recently added to CVC4, allowing string formulas that assert relations such as string equality, concatenation, length, substring, and set membership [8].

Kaluza, a string solver developed as part of the JavaScript symbolic execution framework Kudzu, uses a constraint language that supports regular expressions, length, and concatenation. The Kaluza string solver uses part of the HAMPI implmentation to solve constraints [14].

Similar to CVC4, Z3-str is a project built on top of Microsoft's existing theorem prover, Z3, allowing it to integrate with logic over other datatypes. The authors apply Z3-str to finding remote code execution vulnerabilities [16].

Other existing string solvers take different approaches to defining and solving string constraints. HAMPI solves string constraints primarily by checking for membership of a string in a context-free grammar [7]. For instance, HAMPI allows the user to define regular and context-free languages and assert membership of a string variable in the language. The authors evaluate the tool on PHP programs containing SQL injection vulnerabilities.

Chapter 3

Technical Background

3.1 String-based Security Vulnerabilities

In this project, we focus on the applications of string solving to two types of vulnerabilities: SQL injection and cross-site scripting.

3.1.1 SQL Injection

SQL injections are string-based vulnerabilities where untrusted input manipulates SQL statements submitted to the database. Because an attacker can construct arbitrary queries, this vulnerability gives the attacker power over the database – selecting sensitive data, modifying existing data, or administrating the database [11].

Example: Classical SQL Injection Consider the following example from the OWASP SQL Injection testing page [13].

Suppose the following query is constructed dynamically with variables `$username` and `$password`.

```
SELECT * FROM Users WHERE Username='$username' AND Password='$password'
```

Given a query result set containing multiple users, the server will likely authenticate the user using the first set of matching credentials. Note that if the user provides a username and an *incorrect* password, the resulting query set will be empty. However, suppose instead that a malicious user chooses to provide the following input:

```
$username = "1' or '1' = '1"
$password = "1' or '1' = '1"
```

Let us examine the resulting query by substituting in the values of the variable:

```
SELECT * FROM Users WHERE Username='1' OR '1' = '1'
         AND Password='1' OR '1' = '1'
```

Since `'1' = '1'` is always true, then this SQL statement has the effect of selecting *all* users from the database – authenticating the attacker as the first user in the resulting query set. Additionally, the first user in the database is often the administrative user.

3.1.2 Cross-site scripting

Cross-site scripting (XSS) is a category of web application vulnerabilities where untrusted input allows an attacker to execute arbitrary code on behalf of visiting users to a webpage.

There are two main categories of XSS vulnerabilities: persistent and reflected.

3.1.2.1 Persistent (stored) XSS

In *persistent XSS*, malicious users provide input that is persisted into the database, such that rendering the input allows the malicious user to run an arbitrary script. Since this information is stored in the server and later rendered in the webpage, this allows an attacker to run scripts on the clients of all future users.

Example An example of a persistent XSS vulnerability is unsanitized comments on a blog. Suppose a blog displays a list of comments below each blog post. In the case of a benign user, a comment will likely only contain formatting HTML and text. However, suppose a malicious user submits the following comment:

```
<script type="text/javascript>alert(1);</script>
```

Since the comments use unescaped HTML, future visitors will view the comment, consequently running the script contained in the body. An `alert` is fairly benign, but an attacker could replace the comment body with arbitrary JavaScript.

Most modern blogs protect against such a straightforward exploit, but this example conveys the basic idea behind persistent XSS: an attacker can store information in the database that will be rendered to future visitors of the page, running a potentially malicious script.

3.1.2.2 Reflected XSS

In *reflected XSS*, malicious users provide input in a web request that is later rendered onto the page by the server. Potential vectors for this input include URL parameters, form fields, and cookies. This is called *reflected* XSS, since the input is "reflected" back onto the page by the server's response, such as an error message or user notification.

In contrast with persistent XSS, where the script will be run to all future visitors of a web page, reflected XSS is frequently delivered to victims through a carefully crafted URL.

Example: Query Parameter Consider the example of a search engine that includes the search query in the URL. For example:

```
http://searchengine.com/search.php?q=programming
```

The web page itself will likely display the search query on the user-facing page. However, a malicious user can also craft the following search query:

```
http://searchengine.com/search.php?q=<script>alert(1);</script>
```

The user input is reflected in the contents of the webpage. An attacker could send a URL with more malignant code to victims – potentially masked behind a link shortener – where the code would execute upon visiting the link.

Example: Form field Consider the following example from the OWASP testing page, illustrating the ability to run arbitrary JavaScript without using a `<script>` tag. [12] Consider an HTML form that pre-populates a field with some unsanitized input from the user (`INPUT_FROM_USER`).

```
<input type="text" name="state" value="INPUT_FROM_USER">
```

Suppose an attacker provides the following input:

```
" onfocus="alert(document.cookie)
```

Substituting in the user input, the resulting input field becomes:

```
<input type="text" name="state" value=""
       onfocus="alert(document.cookie)">
```

Operation	Description	Code example
getstatic	Get static field from class	`local = foo.staticf`
putstatic	Set static field in class	`foo.staticf = local`
getfield	Get field in object	`local = foo.f`
putfield	Set field in object	`foo.f = local`
ret/return	Return a local variable or void from a subroutine	`return local`
invokedynamic, ...	Various types of method invocations	`s.toString(); foo.bar()`

TABLE 3.1: Examples of relevant Java bytecode instructions [9]

3.2 Java bytecode

Unlike programming languages like C, where source code compiles directly to assembly, Java source code compiles into *Java bytecode*. Java bytecode serves as platform-independent instructions for the Java virtual machine.

In this project, we perform our analysis on Java bytecode rather than Java source code. We are only concerned with a subset of Java bytecode instructions – in particular, those dealing with fields, variables, and function calls. To give a broad illustration of the functions of bytecode, summaries of the bytecode instructions relevant to this work are detailed below. Each one has an example of the approximate corresponding scenario in Java source code.

3.3 Dataflow analysis

Dataflow analysis describes analysis performed to determine facts about a program at given points in the program. An example of dataflow analysis is liveness analysis, where, for any given point in the program, the analysis computes the set of live variables – variables whose values may be used in the future.

Dataflow analysis uses a *control flow graph* of the program. A control flow graph is a directed graph where each node contains a statement. In this case, each node is an instruction of Java bytecode.

In practice, instructions are represented by an *intermediate representation* of Java bytecode instructions. Intermediate representation refers to a representation of a language using a data structures – useful for dataflow analysis, compilation to a target language, or optimization.

In our analysis, the dataflow analysis is used to compute constraints for mutable string builder variables. This is described in more detail later.

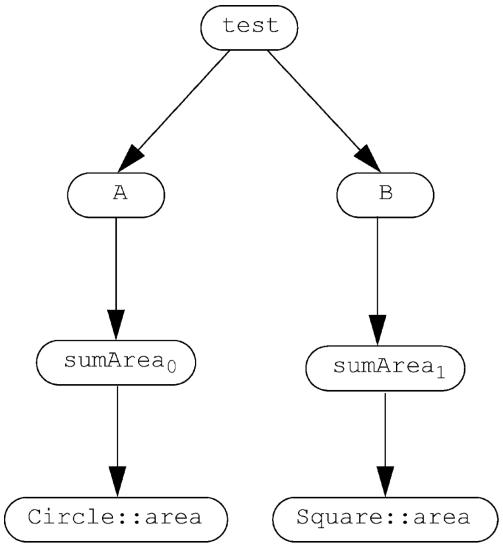

FIGURE 3.1: An example of a context-sensitive call graph (Grove et al.)

3.3.1 Call graphs

Pointer analysis is a type of analysis that determines which memory locations a pointer can point to. In this project, an existing pointer analysis is used to generate a *call graph*. A call graph is a directed graph that captures the relationship between method calls in a program.

A *context-sensitive* call graph distinguishes between different call stacks with which a method may be called. This is more precise than a non-context-sensitive call graph. For example, it differentiates between the case when a same method is called multiple times but with different arguments.

In a context-sensitive analysis, each call graph node consists of two parts: a method and a *calling context*. The contents of a calling context can vary with analysis, but a calling context captures information about a call, potentially distinguishing two different calls of the same method. For example, a simple calling context could contain the line number of the method call, and two calls to a method from different lines of source code would have different calling contexts.

The example below, due to Grove et al. [4], shows a context-sensitive call graph. In the figure, the method `test()` calls `A(), B()`, each of which call the method `sumArea()`. However, the subscripts `0, 1` on the nodes denote different calling contexts of the calls to `sumArea()` – one is called from within `A()`, while the other is called from within `B()`.

3.3.2 Static single assignment

Certain intermediate representations satisfy the property of *static single assignment*. In static single assignment, variables in bytecode can be the left-hand side of an assignment at most once. For example, if the same variable is assigned twice in source code:

```
x := y
u := x
x := z
v := x
```

The resulting SSA representation of these instructions will create new variables for each instance of the variable x.

```
x1 := y
u := x1
x2 := z
v := x2
```

The intermediate representation used in this project is *partial* static single assignment. While local variables follow the constraints of static single assignments, the variables representing the field of an object can have multiple assignments.

3.3.3 Intraprocedural analysis

In *intraprocedural dataflow analysis*, the results of the analysis are derived from a single function. An intraprocedural analysis computes facts by flowing over each bytecode instruction within a method.

When an intraprocedural analysis stands alone, the results of invoking another function are approximated rather than analyzing and using the results of the invoked function. Often, however, an intraprocedural analysis is combined with an interprocedural analysis to compute facts about a whole program.

3.3.4 Interprocedural analysis

While intraprocedural analysis analyzes the contents of a single method, *interprocedural dataflow analysis* accounts for method invocations. Each time a method is invoked in an intraprocedural analysis, an interprocedural analysis framework triggers another *intra*procedural analysis of the invoked method. An interprocedural analysis framework will track the results and compute a fixpoint for facts, handling cases such as recursive functions.

3.4 Satisfiable modulo theory

Satisfiable modulo theories (SMT) are a type of decision problem – a problem that returns a yes or no answer. SMT generalizes the Boolean satifiability problem (SAT).

In the SAT decision problem, one must determine whether a boolean formula has a satisfying assignment. For example, the formula:

$$(x \wedge y) \vee (y \wedge \neg z)$$

has the following satisfying assignment:

$$x = \text{true}, y = \text{true}, z = \text{false}$$

In contrast, SMT generalizes SAT. Boolean variables can be replaced with other predicates, expanding the formulas beyond boolean variables. Each of these types are referred to as *theories*. An SMT problem can incorporate the theory of real numbers, the theory of lists, and so on. In this work, we are particularly interested in SMT problems employing the theory of strings – solving formulas with string predicates.

3.4.1 String solvers

The term *string solver* will be used throughout this work, referring specifically to SMT solvers with the capability of solving string formulas. Just as SAT solvers find satisfying boolean assignments for assertions, string solvers find satisfying string assignments.

Consider the following example with string predicates. If we have the following assertions:

$$(s = \langle \text{any string} \rangle) \wedge (r = \text{“}bar\text{”}) \wedge (t = concat(s, t)) \wedge (t \text{ begins with “}foo\text{”})$$

A solution that satisfies these formulas would be:

$$s = \text{“}foo\text{”}, r = \text{“}bar\text{”}, t = \text{“}foobar\text{”}$$

However, note that the following would also be a solution:

$$s = \text{“}fooooooooooaa\text{”}, r = \text{“}bar\text{”}, t = \text{“}foobar\text{”}$$

Given the original formula, a string solver would either return at least one solution or indicate that no solution exists.

3.5 Contribution

The primary contribution of this work is a tool that leverages an existing string solver for solving string constraints for Java bytecode programs, illustrating the feasibility of an end-to-end pipeline for generating and solving string constraints. We built a tool that achieves this through the following components:

1. A dataflow analysis that generates string constraints from Java programs
2. A translation from string constraints to the language of SMT solvers

Chapter 4

Design

In this section, we describe the design of the analysis that generates string constraints from a given Java program.

4.1 Concepts

4.1.1 String Variables

In this project, a *string variable* represents a string manipulated or computed by the program. Concretely, this encompasses immutable Java `String` or mutable `StringBuilder` types. String variables encapsulate two types of variables within a program:

1. *Program variables*, which represent local variables or object fields.
2. *Formal variables*, which represent the formal arguments or return values of a method.

Ultimately, the purpose of string variables is to determine the string values that a string variable can take on in the input program. Given string variables, we will declare constraints such as "the string value that `sv1` takes on is equal to that of `sv2`". The domain of constraints is described in section 4.1.2.

4.1.1.1 Local variables

A *local variable* refers to any variable declared within a method.

In the example source code below, a string variable would be associated with each of the variables `foo, bar`:

```
public void myMethod(String a) {
    String foo = a;
    String bar = String.concat(a, a);
}
```

4.1.1.2 Field variables

A *field variable* is a type of program variable representing a field in a Java object.

In the example source code below, both `f0, f1` would be represented in the analysis by field variables. String literals are also represented by string variables, and in the example below, the string `"foo"` would be represented by a third string variable.

```
public class Foo {
    private String f0;
    private static String f1 = "foo";

    // ...
}
```

4.1.1.3 Formal variables

Formal variables are used to summarize information about methods. A formal variable corresponds to either a method argument or a method return value. The high-level concepts are described below, and its use is described in further detail in section 4.2.2.1 on summary nodes.

In the case of method arguments, a formal string variable is associated with each string parameter of a method and its call graph node. For example, given two call graph nodes for the same method, each will have a different formal string variable associated with the same string-typed method parameter.

Consider the following method with string-valued arguments:

```
public void myMethod(String arg0, int arg1, String arg2)  {
    // ...
}
```

There will be a formal string variable associated with the string-valued arguments `arg0` and `arg2`.

In the case of return values, a formal string variable is used to represent the value returned by a method. A string method will return some program variable. This program

variable will be associated with the formal return variable for the method. This association is described in more detail in section 4.2.2. This applies only to methods that return a string type, e.g.:

```
public String anotherMethod() {
    // ...
}
```

4.1.2 String Constraints

String constraints are used to describe the relationship between string variables. They capture information about the string values that a string variable can take on in the program.

The domain of string constraints generated by this analysis is described below.

We first define the following domains:

- $v \in \textbf{Var}$: string variables
- $s \in \textbf{String}$: string literals (e.g., “foo”)
- $m \in \textbf{CGNode}$: call graph nodes

Let σ be a solution to the constraints, having the domain:

$$\sigma : \textbf{Var} \rightarrow \textbf{String}$$

σ maps string variables to string literals. For example, an explicit instance of σ would be:

$$\sigma_0 = \{v_1 \mapsto \text{"}foo\text{"}; v_2 \mapsto \text{"}bar\text{"}\}$$

For some constraint C, we say that the solution σ *satisfies the constraint* C if:

$$\sigma \vDash C$$

The Table 4.1 defines the domain of constraints on the left side. On the right side, it defines how a solution σ satisfies the constraint.

$v = s$	[Constant]	$\sigma \vDash v = S \iff \sigma(v) = S$
$v_1 = v_2 + +v_3$	[Concat]	$\sigma \vDash v_1 = v_2 + +v_3 \iff \sigma(v_1) = \sigma(v_2) + +\sigma(v_3)$
$v_1 = v_2$	[Copy]	$\sigma \vDash v_1 = v_2 \iff \sigma(v_1) = \sigma(v_2)$
$v = \phi(v_1, \ldots, v_n)$	[Phi]	$\sigma \vDash v = \phi(v_1, \ldots, v_n) \iff \exists i \in \{1, \ldots n\}.\ \sigma(v) = \sigma(v_i)$
$m(v_1^1, \ldots v_n^1) \vee \cdots \vee m(v_1^m, \ldots v_n^m)$	[Call]	$\sigma \vDash m(v_1^1, \ldots v_n^1) \vee \cdots \vee m(v_1^m, \ldots v_n^m) \iff \exists i \in \{1, \ldots, m\}.\ \sigma(v_1^i) = f_1 \wedge \cdots \wedge \sigma(v_n^i) = f_i$ where $formals(m) = (f_1, \ldots, f_n)$

Table 4.1: Summary of the constraints used in this project

4.2 Constraint Generation

Our string constraint analysis generates string constraints using an interprocedural dataflow analysis on the control flow graph of the program. These constraints are later translated into the language of the string solver, which finds a satisfying solution for the constraints.

String constraints are primarily generated from the program by doing one pass over the control flow graph. However, multiple passes may be required for mutable strings. In order to handle mutable strings (`StringBuilder`), we use dataflow analysis to track string variables corresponding to `StringBuilder` at some point in the program. This is further described below in the section on string builders.

4.2.1 Intraprocedural Analysis

In the intraprocedural analysis, we handle cases for each possible Java bytecode instruction, potentially generating a string constraint.

First, we describe the simpler cases. An assignment from a field to a local variable (or vice versa) generates a Copy constraint. When we encounter a phi node, a Phi constraint is generated.

We describe constraint generation for library methods and string builders below.

4.2.1.1 String Library Methods

When we encounter a method invocation, we first check if it is a method in the `String` library that is specially supported by the analysis. If so, we ignore the standard interprocedural analysis and generate specific constraints. The following `String` library methods are supported:

- `new String(s)` (constructor)
- `String.valueOf(s)` (string representation)

- `s.toString()` (to string)
- `String.concat(s, t)` (concatenation)

The first three generate a `Copy` constraint. For example, the bytecode corresponding to the constructor method invocation of the code below:

```
String s = new String(t);
```

will generate the following constraint:

$$s = t \qquad [\textsc{Copy}]$$

Similarly, a call to `String.concat(s, t)` will generate a Concat constraint.

4.2.1.2 String Builders

In Java, string builders are a class for constructing mutable strings. In order to handle string builder manipulations, we create a new string variable each time a string builder is manipulated.

By creating a new string variable, we represent a snapshot of the mutable string builder at a certain program point. The dataflow analysis computes a *variable context*: a mapping from program variables to string variables. After a string builder program variable is changed, while the program variable may have been mutated, the context tracks the most recent string variable associated with the string builder.

In a node of the control flow graph with multiple incoming nodes, we compute the confluence and merge the incoming contexts by creating Phi constraints for string variables associated with the same program variable.

Example Consider the following example below, where two strings, `s1, s2`, are appended to the string variable `sb`.

```
StringBuilder sb = new StringBuilder("");
sb.append(s1);   // new string variable created - sb1
sb.append(s2);   // new string variable created - sb2
```

Each time that `sb` is manipulated by appending another string, a new string variable is created. The current context computed by the dataflow analysis is updated, associating the most recent string variable `sb2` (generated by `sb.append(s2)`) with the program variable `sb`. After this code executes, if we were to look up the string variable associated with `sb`, then we would find the string variable `sb2`.

String Builder Library Methods The following `StringBuilder` (and the synchronized equivalent, `StringBuffer`) library methods are supported:

- `new StringBuilder(s)` (constructor)
- `sb.toString(sb)` (to string)
- `sb.append(s)` (append)

More importantly, note that the Java syntactic convenience of appending strings with `+` is compiled in bytecode to corresponding calls to `StringBuilder.append()`:

```
String s = t + v;  // t, v are String variables
String fb = "foo" + "bar";
String newS = s + "foo";
```

Our implementation handles this syntax, which is frequently used by programmers to construct strings.

4.2.2 Interprocedural Analysis

We describe how the interprocedural analysis handles method invocations. First, we describe the concept of *summary nodes*, then we describe how these are used to generate the CALL constraint.

4.2.2.1 Summary Nodes

Summary nodes contain information about the string variables associated with a call graph node. Their purpose is to capture information about method calls, allowing us to generate string constraints in an interprocedural analysis.

Recall that a call graph node consists of two parts: a method and a calling context. There exists a one-to-one correspondence between call graph nodes and summary nodes.

Intuitively, a summary node contains information about the string variables of a single method. Note that new variables are generated for each calling context – two different calling contexts have different formal variables.

Formally, a summary node consists of:

1. A list of formal parameter variables $f_1, \ldots, f_n$ for each string-typed parameter
2. A formal return variable r_m (if the method returns a string type)

4.2.2.2 Handling formals

A method has a set of formal arguments, corresponding to the parameters of the method. When a method is called, a series of actual arguments are supplied to the method. In this section, we describe the process of generating constraints that link the actual arguments to the formal arguments.

Suppose that the two calls below to the method `bar()` have the same calling context – that is, the calls will correspond to the same call graph nodes.

```
private String foo() {
    String firstCall = bar("a", "b");
    String secondCall = bar("c", "d");
}
```

The following constraint is generated, supposing there exists some string variables a, b, c, d that represent the literals `"a", "b", "c", "d"`:

$$bar(a, b) \vee bar(c, d)$$

Note that this constraint is only satisfied by a solution σ when, for the formal variables f_1, f_2 of `bar()`:

$$\sigma \vDash bar(a, b) \vee bar(c, d) \iff (\sigma(a) = f_1 \ \wedge \ \sigma(b) = f_2) \vee (\sigma(c) = f_1 \ \wedge \ \sigma(d) = f_2)$$

This captures the idea that the actual arguments to a method should be "grouped" together for a more precise analysis. The possible two arguments provided to `bar()` include the argument set `("a", "b")` and `("c", "d")`, but note that our approach to generating constraints disallows the combination `("a", "d")`, since `bar()` is never called with this combination of arguments.

In this example, we assume that both calls have the same calling context and consequently the same summary node. Note that this disjunction of argument sets is grouped together by summary node (and thus, call graph node) and not method. The constraints will distinguish argument sets between calls to the same method with different calling contexts.

4.2.2.3 Handling return

The bytecode instruction for return takes a program variable as an argument – the variable to be returned. When we flow over a return statement in bytecode, we add the returned variable to a set of tracked variables.

After analyzing a method, we add a new PHI constraint to link the formal return variable to the possible tracked variables. For a given summary node, we generate the constraint:

$$m_r = \phi(r_1, \ldots, r_n) \qquad \text{[PHI]}$$

where $r_1, \ldots, r_n$ are the variables potentially returned by the method at some point in its execution.

4.2.2.4 Example

We describe an example of generating constraints in a very basic program. The program calls a method, `addBar()`, that appends the string literal "bar" to its argument. Consider the following program:

```
public class InterproceduralExample {
    static final String s = "foo";

    public static void main(String[] args) {
        String s1 = s;
        String s2 = addBar(s1);  // should be "foobar"
    }

    private static String addBar(String arg) {
        String local = "bar";
        return arg.concat(local);
    }
}
```

We walk through the example line-by-line, using names to denote the locations of string variables (e.g., *main-literal-foo* refers to a string variable in `main()` representing the literal "foo"). Our analysis proceeds by method.

Analyzing `main()` When we access a static field:

```
String s1 = s;
```

it generates the constraint:

$$\textit{main-literal-foo} = \text{“}foo\text{”} \qquad \text{[CONSTANT]}$$

When we make a call to `addBar(s1)`:

```
String s2 = addBar(s1);  // should be "foobar"
```

we find the summary node for the call graph node corresponding to the call, and we link the actual variable (*main-literal-foo*, which represents `s1` in the program) to the formal variable. We also link the return variable of `addBar()` to the string variable corresponding to `s2`. This generates the constraint:

$$addBar(main\text{-}literal\text{-}foo) \qquad [\textsc{Call}]$$
$$main\text{-}local\text{-}s_2 = addBar\text{-}return \qquad [\textsc{Copy}]$$

Analyzing `addBar()` Our interprocedural analysis now delves into the invoked method, performing an intraprocedural analysis on `addBar()`.

We link the formal variables to the corresponding local arguments. In bytecode, we see that `addBar()` is defined with a parameter named `arg`, which acts as a local variable:

```
private static String addBar(String arg) { ...
```

This generates the constraint:

$$addBar\text{-}f_0 = addBar\text{-}local\text{-}arg \qquad [\textsc{Copy}]$$

As we analyze the body of `addBar()`, we encounter a string literal assignment:

```
String local = "bar";
```

generating the constraint:

$$addBar\text{-}literal\text{-}bar = \text{“}bar\text{”} \qquad [\textsc{Constant}]$$

Finally, we want to return the result of concatenating the string literal:

```
return arg.concat(local);
```

Note that the bytecode implicitly creates a new local variable v_0 in the bytecode, not corresponding to any variable in the source code. This first generates the constraint:

$$addBar\text{-}local\text{-}v_0 = addBar\text{-}local\text{-}arg \mathbin{+\!\!+} addBar\text{-}literal\text{-}bar \qquad [\textsc{Concat}]$$

We link the formal return variable using a Phi constraint of all of the possible local variables that can be returned. In this case, there is only one, meaning that the return statement generates the following constraint (equivalent to a `Copy` constraint):

$$addBar\text{-}return = phi(addBar\text{-}local\text{-}v_0) \qquad [\textsc{Phi}]$$

Solution These constraints are translated to the language of the string solver, which solves for a satisfying assignment to all of the variables. Since this program is simple, we have an intuition that our solver should find that the variable *main-local-*s_2 will be "foobar".

Consider the following solution. By inspection, it can be confirmed that this satisfies the constraints.

$$\begin{aligned}\sigma = [&\textit{addBar-}f_0 \mapsto \text{“}foo\text{”}\\ &\textit{main-literal-}foo \mapsto \text{“}foo\text{”}\\ &\textit{addBar-local-}arg \mapsto \text{“}foo\text{”}\\ &\textit{addBar-literal-bar} \mapsto \text{“}bar\text{”}\\ &\textit{addBar-local-}v_0 \mapsto \text{“}foobar\text{”}\\ &\textit{addBar-return} \mapsto \text{“}foobar\text{”}\\ &\textit{main-local-}s_2 \mapsto \text{“}foobar\text{”}]\end{aligned}$$

4.3 Limitations

When our tool analyzes for security vulnerabilities, we will add more constraints relating to the variable of interest. For example, in the case of SQL injection, this is the string that is sent to the database as a query. This is further elaborated in the section on evaluation.

The analysis is neither sound nor complete. If the string solver does not find a satisfying solution, this does not guarantee that the program is free of vulnerabilities. If the string solver finds a satisfying solution, this does not guarantee that the program contains a vulnerability.

Our analysis is useful as a vulnerability finding tool, which is demonstrated in its evaluation in the next section. If the string solver returns a satisfying solution, it provides a starting point for the programmer to find potential security flaws.

There are three notable limitations to our analysis.

First, the analysis only generates constraints for a few major string operations, such as concatenation and assignment. It excludes many elementary operations that manipulate strings – such as substring replacement, regular expression replacement, and character replacement – due to the complexity of these analyses. These operations are useful in a precise analysis, since many security related functions, such as sanitizers, employ

such manipulations. For example, our analysis currently cannot analyze a SQL injection sanitizer that escapes single quotations, substituting ' for \'.

Second, the analysis does not support full-fledged web applications. Performing static analysis on the frameworks and XML configurations that most Java web frameworks use is complex, illustrated by projects such as Framework 4 Frameworks [15]. Our evaluation is performed on toy Java applications that simulate features of web application frameworks, such as web requests and database connections.

Third, due to the limitations in capabilities of available string solvers, a satisfying solution binds a string variable to exactly one string. In contrast, there are alternative ways of expressing the set of strings that a string variable can take, such as associating each string variable with a context-free grammar approximating a set of strings. This leads to certain limitations in the analysis. For example, in a loop, where a variable takes on different values during through iterations of the loop, our ability to represent such variables is severely limited.

Chapter 5

Implementation

5.1 Constraint Analysis

The analysis was developed using Accrue Bytecode, an existing framework for Java bytecode analysis. Accrue Bytecode leverages the IBM T.J. Watson Libraries for Analysis (WALA) framework for representing Java bytecode.

The analysis and translation were written in Java.

5.2 SMT Solver

The CVC4 SMT solver was used to solve string constraints. CVC4 was selected over other existing string solvers, such as Kaluza [14], HAMPI [7], and Z3-str [16], for its expanded support of string formulas. The constraints were solved using the CVC4 Java API bindings.

5.3 Evaluation

The tool was run on example programs using a 2 GHz Intel Core i7 Macbook Pro with 8 GB of RAM.

Chapter 6

Evaluation

The constraints that we generate provide a foundation for approximating the string values that variables take on at given program points. In this section, we demonstrate how our analysis can be augmented to detect security vulnerabilities in Java programs by adding certain constraints.

To evaluate our end-to-end tool for generating and solving constraints for a given program, we tested it on two example programs. These example programs contain SQL injection or cross-site scripting vulnerabilities.

6.1 Example programs

6.1.1 SQL Injection

We evaluated the tool on a 43 line example Java program, SQLToyApp, that simulates the login of a website. It accepts a username and password as input, sending the pair of strings to the database to validate. The full code is included in the appendix.

Given user inputs `$username`, `$password`, `SQLToyApp` sends the following query:

```
SELECT * FROM users WHERE username = '$username'
         AND password = '$password'
```

Our strategy for determining whether the exists a SQL injection follows. Following the model of evaluation for the HAMPI string solver, we test whether there is a satisfying assignment of strings with the query string containing the substring `"1' or '1' = '1"`. [7] To achieve this, we add the following assertion in the string solver, using the

string variable q for the query string sent to the database.

q contains "1' or '1' = '1"

We fix the input for $username to be "admin", assuming that the attacker aims to login a username for the administrative account. However, note that an attacker could also input username = "'1' OR '1' = '1'", which would return a query set containing all users – the first of which would be authenticated in the login.

This is motivated by the fact that the most common SQL injection attacks take advantage of unescaped quotation and tautologies (or '1' = '1'). More complex SQL injection vectors, such as those with stored procedures, would require a different approach.

Results Our tool ran the pointer analyses, generated 30 string constraints, and discovered a satisfying solution containing a vulnerability in 9.054 seconds. The satisfying solution contained 31 string variables. Of particular interest is the assignment for the query variable q, for which the string solver solution finds that:

```
q = SELECT * FROM users WHERE username = 'admin'
              AND password = '1' or '1' = '1'
```

This corresponds to the scenario where a malicious user provides the following inputs: $username = 'admin' and $password = 1' or '1' = '1.

This illustrates that the tool produces a string assignment that highlights a SQL injection vulnerability in SQLToyApp. Given a Java program as input and additional constraints designating the strings of interest, our tool outputs a satisfying solution that indicates a SQL injection vulnerability.

6.1.2 Cross-site scripting

We evaluated the tool on a 32 line example Java program, XSSToyApp, that simulates the canonical example of a blog post, rendering an HTML page that takes a user comment as input. The full code is included in the appendix.

Recall that if user comments in a blog are unescaped, then a malicious user can simply provide <script> tags containing arbitrary code that will be executed by any future clients viewing the comments. Our strategy for finding an XSS vulnerability is to add

the following assertion to the string solver, where `html` is the rendered HTML of the final webpage:

`html` contains `"<script>alert(1);</script>"`

where `alert(1)` serves as an arbitrary choice of JavaScript code. A malicious user would replace this with something more harmful.

Results Our tool ran the pointer analyses, generated 24 string constraints, and discovered a satisfying solution containing a vulnerability in 8.001 seconds. The satisfying solution for the tool assigns the following value to the string variable for the rendered HTML:

```
html = "<!DOCTYPE html>
            <html>
                <body>
                    <div><script>alert(1);</script></div>
                </body>
            </html>"
```

This corresponds to the scenario where a malicious user writes the following comment:

```
<script>alert(1);</script>
```

Similar to the SQL injection example, this applies the tool to a program representative of the canonical persistent XSS vulnerability.

Chapter 7

Conclusion

We designed a tool to detect security vulnerabilities in Java web applications, evaluating the tool on simulated web applications.

First, the tool generated string constraints from a Java program using interprocedural dataflow analysis. Second, it translated the string constraints to the language of the CVC4 string solver. Finally, the string solver generated a satisfying assignment of string variables from the given constraints.

We evaluated the tool by demonstrating that it could find satisfying assignments indicative of security vulnerabilities – SQL injection and cross-site scripting – in example Java programs.

However, the tool was subject to certain limitations. Although the analysis was neither sound nor complete, the tool can be used to guide programmers to potential security vulnerabilities.

This work provides the foundation for an end-to-end tool that generates string constraints and pipes them into an SMT solver. Future work in the area could expand the tool to support the infrastructure of Java web frameworks and further string library operations, which would allow analysis of sanitizing functions commonly used to secure applications.

Finally, the goal of leveraging an existing string solver highlights the limitations of existing string solvers, which often return a satisfying assignment that assigns each string variable to a single string value. String solvers with more expressiveness for solutions, such as associating each string variable with a context-free grammar for the set of possible strings, would allow for a more powerful analysis.

Appendices

Appendix A

Example programs

Below are the example programs, SQLToyApp, XSSToyApp, used in the evaluation.

A.1 SQL Injection

```
package stringconstraint.tests;

import java.lang.StringBuilder;

/**
 * An application that receives a username and a password from a form field
 */
public class SQLToyApp {
    public static void main(String args[]) {
        SQLToyApp app = new SQLToyApp();
        app.login(args[0], args[1]);
    }

    public void login(String username, String password) {
        String query = constructQuery(username, password);
        DatabaseConnection dbc = new DatabaseConnection();
        dbc.sendQuery(query);
    }

    private String constructQuery(String username, String password) {
        StringBuilder query = new StringBuilder("SELECT * FROM users WHERE ");
        query.append("username = '");
        query.append(username);
        query.append("'");
```

```
        query.append(" AND ");
        query.append("password = '");
        query.append(password);
        query.append("'");
        query.append(";");
        return query.toString();
    }

    /**
     * Mock database connection.
     */
    public class DatabaseConnection {
        public void sendQuery(String q) {
            // Do nothing.
        }
    }
}
```

Listing A.1: SQLToyApp, a program simulating a query for a user login

A.2 XSS

```
package stringconstraint.tests;

import java.lang.StringBuilder;

/**
 * An example application that renders an HTML page containing
 * an unescaped comment from a user.
 */
public class XSSToyApp {
    public static void main(String args[]) {
        // Suppose the comment, args[0], is retrieved from a database elsewhere
        XSSToyApp app = new XSSToyApp();
        app.renderPage(args[0]);
    }

    private String buildPage(String comment) {
        StringBuilder sb = new StringBuilder("<!DOCTYPE html>");
        sb.append("<html>");
            sb.append("<body>");
                sb.append("<div>");
                    sb.append(comment);
```

```
                sb.append("</div>");
            sb.append("</body>");
        sb.append("</html>");
        return sb.toString();
    }

    public void renderPage(String comment) {
        String html = buildPage(comment);
        // Do something with the built page.
    }
}
```

Listing A.2: `XSSToyApp`, a program simulating the rendering of a webpage

References

[1] Clark Barrett, Christopher L Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. Cvc4. In *Computer aided verification*, pages 171–177. Springer, 2011.

[2] Aske Simon Christensen, Anders Møller, and Michael I Schwartzbach. *Precise analysis of string expressions*. Springer, 2003.

[3] Xiang Fu and Chung-Chih Li. A string constraint solver for detecting web application vulnerability. In *SEKE*, pages 535–542, 2010.

[4] David Grove, Greg DeFouw, Jeffrey Dean, and Craig Chambers. Call graph construction in object-oriented languages. *ACM SIGPLAN Notices*, 32(10):108–124, 1997.

[5] William GJ Halfond and Alessandro Orso. Amnesia: analysis and monitoring for neutralizing sql-injection attacks. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pages 174–183. ACM, 2005.

[6] Pieter Hooimeijer, Benjamin Livshits, David Molnar, Prateek Saxena, and Margus Veanes. Fast and precise sanitizer analysis with bek. In *Proceedings of the 20th USENIX conference on Security*, pages 1–1. USENIX Association, 2011.

[7] Adam Kiezun, Vijay Ganesh, Philip J Guo, Pieter Hooimeijer, and Michael D Ernst. Hampi: a solver for string constraints. In *Proceedings of the eighteenth international symposium on Software testing and analysis*, pages 105–116. ACM, 2009.

[8] Tianyi Liang, Andrew Reynolds, Cesare Tinelli, Clark Barrett, and Morgan Deters. A dpll (t) theory solver for a theory of strings and regular expressions. In *Computer Aided Verification*, pages 646–662. Springer, 2014.

[9] Oracle. Chapter 6. the java virtual machine instruction set, 2015.

[10] Open Web Application Security Project. Top 10 2013-top 10, 2013.

[11] Open Web Application Security Project. Sql injection, 2015.

[12] Open Web Application Security Project. Testing for reflected cross site scripting (otg-inpval-001), 2015.

[13] Open Web Application Security Project. Testing for sql injection (otg-inpval-005), 2015.

[14] Prateek Saxena, Devdatta Akhawe, Steve Hanna, Feng Mao, Stephen McCamant, and Dawn Song. A symbolic execution framework for javascript. In *Security and Privacy (SP), 2010 IEEE Symposium on*, pages 513–528. IEEE, 2010.

[15] Manu Sridharan, Shay Artzi, Marco Pistoia, Salvatore Guarnieri, Omer Tripp, and Ryan Berg. F4f: taint analysis of framework-based web applications. *ACM SIGPLAN Notices*, 46(10):1053–1068, 2011.

[16] Yunhui Zheng, Xiangyu Zhang, and Vijay Ganesh. Z3-str: A z3-based string solver for web application analysis. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 114–124. ACM, 2013.