Abstracting Symbolic Execution with String Analysis

Daryl Shannon
dshannon@ece.utexas.edu

Sukant Hajra
tnks@cs.utexas.edu

Alison Lee
alison_lee@mail.utexas.edu

Daiqian Zhan
zhan@ece.utexas.edu

Sarfraz Khurshid
khurshid@ece.utexas.edu

The University of Texas at Austin
Austin, TX 78712

Abstract

Forward symbolic execution is a technique for program analysis that explores the execution paths of a program by maintaining a symbolic representation of the program state. Traditionally, applications of this technique have focused on symbolically representing only primitive data types, while more recent extensions have expanded to reference types. We demonstrate the ability to symbolically execute a commonly used library class, specifically a string, at an abstract level. By abstracting away the implementation details of strings using finite-state automata, symbolic execution can scale to more complex programs. This technique can be applied to programs which generate complicated strings, such as SQL database queries.

1. Introduction

Over the last three decades, symbolic execution has been widely explored in the areas of software testing and verification [1,3,6,8,14,16,18–21]. Early implementations of symbolic execution mostly dealt with primitive types, such as integers and arithmetic operations. Recent development has generalized symbolic execution to handle references and array types. Still, prior to our work symbolic execution has been performed at the representation level (primitives, arrays, and reference types). We believe a key obstacle to the scalability of symbolic execution is performing it at such a level. Indeed if the *entire* implementation needs to be checked, such execution is necessary. However, it is redundant if the aim is to only check the client code (trusting the correctness of the implementation of selected libraries).

For example, consider as a simple illustration, a program that uses a library implementation of the abstract data type `Set`. Suppose the method adds an element `e` to a set `S`, and then checks whether `e` belongs to `S`. Executing this program using representation level manipulations involves executing methods that implement the add and membership check operations. For performance reasons, sets are typically implemented using complex data structures, such as balanced binary search trees, which involve complex (balancing) operations. Symbolically executing these operations rapidly generates large path conditions which quickly lead to infeasible analyses.

By abstracting out the implementation details of common library classes, we can reduce the complexity of a program and enhance the scalability of symbolic execution. Based on our previous work [15], we have developed in Java a prototype, named Haderach, that symbolically executes over the `java.lang.String` class. We present results from experiments using a variety of programs that make extensive use of strings. These tests demonstrate the effectiveness of our implementation. Our abstraction technique can be extended beyond strings to other classes and libraries as well. For instance, we may symbolically execute over standard container classes such as lists, trees, or maps.

Background information on symbolic execution is presented in Section 2. In Section 3, we explain our novel technique for symbolic execution. Section 4 describes the specifics of our prototype implementation. Section 5 presents our test results, and Section 6 discusses future work and conclusions.

This paper introduces the idea of performing String analysis using symbolic execution and using automota-based representations for abstracting strings during symbolic execution. Our prototype is under development and at present handles a few core methods in `String`, `StringBuffer` and `StringBuilder` classes from the standard Java libraries. We describe our approach using small but illustrative programs that perform commonly used operations on Strings.

2. Background

Forward symbolic execution is a technique for executing a program on symbolic values [16]. There are two fundamental aspects of symbolic execution: (1) defining symbolic semantics for operations that are originally defined for concrete values and (2) maintaining a *path condition* for the current program path being executed. A path condition specifies necessary symbolic constraints on input variables that must be satisfied to execute the corresponding path.

As an example, consider the function in Listing 1 that returns the absolute value of its input. To symbolically execute this program, we consider its behavior when the input variable `i` contains a symbolic value, say I_0. We make no assumptions about the value of I_0 (except for what we deduce from the `int` type declaration). So when we encounter a conditional statement, we consider both possible outcomes of the condition. To perform operations on symbols, we treat them simply as variables. For example, the statement on line L3 would update the value of `result` to be $-1 * I_0$. Thus a symbolic executor must also manage `result` symbolicly to support such updates.

Symbolic execution of the code in Listing 1 explores the following two paths:

Path 1: $[I_0 < 0]\ L1 \to L2 \to L3 \to L5$

Path 2: $[I_0 \geq 0]\ L1 \to L2 \to L4 \to L5$

Note that for each path explored, there is a corresponding path condition (shown in square brackets). While execution on a concrete input would have followed exactly one of these two paths, symbolic execution explores both.

To derive meaningful results from the path condition symbolic execution tools require a constraint solver that can either (1) solve for the satisfiability of the path condition or (2) generate a *witness* for the path condition, a concrete example input that satisfies the path condition. Solving for the satisfiability of a path condition allows the symbolic executor to improve its performance by ignoring impossible paths. Additionally, a symbolic executor can inject specialized constraints at user-specified "hotspots" in a program and solve for satisfiability to establish program invariance. If a program is found to potentially violate an invariant, then a constraint solver supporting witness generation can produce a counter-example. Also, witness generation, when applied to all path conditions, can produce a compact set of test cases with broad code coverage.

3. Our Approach

As stated in the background section, there are two fundamental aspects of symbolic execution: defining symbolic semantics for operations and maintaining a path condition.

Listing 1. An absolute value function

```
int abs(int i) {
    int result;             // L1
    if (i < 0)              // L2
        result = -1 * i;    // L3
    else result = i;        // L4
    return result;          // L5
}
```

Traditionally, the path condition is stored as a list of constraints on the programs inputs. Our technique involves taking a different approach for representing constraints on strings. Taking inspiration from the Java String Analyzer (JSA) [5], we use finite-state automata for symbolic string values. Each symbolic value's automaton accepts all strings which satisfy the path condition. In this way constraints on a symbolic string value are represented in its automaton. When a new string constraint is conjoined to the path condition, instead of adding it to the list, we can under certain circumstances refine the automaton so that it no longer accepts strings that conflict with the constraint. When the path condition is satisfied, any string accepted by the automaton can be chosen as a witness for that path. If a refined automaton does not accept any strings then the path condition is no longer satisfiable. Due to this, the automaton is both the data storage for conditions on symbolic string values and the constraint solver for those conditions.

In order to define symbolic semantics for operations on string values we will examine three types of string operations: those that check a predicate for a string (Section 3.1), those that return a primitive (Section 3.2), and those that output a string (Section 3.3).

3.1. String Predicates

Operations that check a predicate for a string usually compare one string with a second string, such as the `equals()` or `startsWith()` methods for Java. These return a boolean value signifying whether or not the predicate holds.

As King described symbolic execution in [16], a predicate q is evaluated using the following expressions, where pc is the path condition:

$$pc \to q \tag{1}$$

$$pc \to \neg\, q \tag{2}$$

If expression (1) evaluates to $true$ then q is $true$. If expression (2) is $true$ then q is $false$. If neither expression (1) nor expression (2) is $true$ then we do not know whether or not q is $true$ and execution must fork into two different paths.

For one path, we decide that q is $true$ and add it to the path condition. For the other path, we decide that q is $false$ and add $\neg\, q$ to the path condition.

When checking string predicates in certain situations, we can evaluate these expressions and add constraints to the path condition by just refining the automata. To illustrate this process we will consider the automata shown in Figure 1 to check the predicate `x.equals(y)`. Assume that `x` contains the automaton illustrated in Figure 1(a) and `y` contains the automaton in Figure 1(b). This means that `y` is "`abc`" and `x` could be "`abc`", "`pqr`", or "`xyz`".

To evaluate the expressions (1) and (2) we will use a proof by contradiction. We should refine the input automata so that our predicate is $false$. If either refined automaton accepts no strings then there is a contradiction and the predicate must follow from the path condition making expression (1) $true$. In our example, comparing the equality of a known string `y` and an unknown string `x`, we should refine `x` to be the intersection of `y`'s complement and `x`. So, `y` would remain Figure 1(b), and `x` would be refined to the automaton in Figure 1(c). Neither `x` nor `y`'s automaton is empty so expression (1) is $false$. For expression (2) we should refine the automata so that our predicate is $true$. We refine `x` to be the intersection of `x` and `y`, resulting in both `x` and `y` containing the automaton in Figure 1(b). Again, neither automaton is empty so expression (2) is also $false$.

Since both expressions are $false$ we will need to fork into two paths. For the first path we should add the predicate to the path condition. This can be done by refining the automata so that the predicate is $true$, just as we did when evaluating expression (2). This results in both `x` and `y` being "`abc`". For the other path, where we add the negation of the predicate to the path condition, we refine the automata so that the predicate is $false$, like expression (1). Here `y` is still "`abc`", but `x` is either "`xyz`" or "`pqr`". The algorithms for these refinements are specific to the predicate being checked. They should be minimal so that all strings which were accepted by the unrefined automaton and do not conflict with the new constraint are accepted by the refined automaton. The union of the strings accepted by the refined automata from each path should be the same strings accepted by the unrefined automaton before branching. Looking at Figure 1, we can see that Figure 1(a) is the union of Figure 1(b) and Figure 1(c).

Sometimes we cannot add a constraint on a symbolic string value to the path condition by just refining the string's automaton. For instance, consider again `x.equals(y)` where `x` is symbolically represented by Figure 1(a), but this time where `y` is symbolically represented by Figure 1(d). In this case `y` has two options, "`abc`" and "`xyz`". For the path in which we add `x.equals(y)` to the path condition, we can only limit `x` to "`abc`" or "`xyz`", the same possibilities as `y`. More specifically, `x` and `y` should both be refined down to their intersection, also Figure 1(d). This alone, is not sufficient to ensure `x` and `y` are equal in this path, as seen in the following code segment:

```
if (x.equals(y))
  if (x.equals("abc"))
    if (y.equals("xyz"))
      // this point is unreachable
```

Once again consider that Figure 1(a) represents `x` and Figure 1(d) represents `y` as we enter the code. If we examine the path where all the `if` statements are true, `x` will be refined to Figure 1(d) after the first line. At the second, x will be further refined to "`abc`", shown in Figure 1(b). The third line will refine `y` to "`xyz`". Clearly "`abc`" is not equal to "`xyz`", so we have a contradiction. When `x` is refined on line 2, we should also refine `y`, because its value depends on `x`. This *symbolic value dependency* must be accounted for somewhere.

In the simple case of `x.equals(y)` we could just make `x` and `y` use the same automaton in memory, so refinements to one will be reflected in the other. However, many predicates, like the `startsWith()` method, require a more complex solution. This can be done in several ways. One option is to further fork the execution so that the automata represent a portion of the possible strings. This results in multiple forks of execution with different path conditions representing all the possibilities of a single path of execution. Clearly, automata that accept a large number of strings may not be tractable with this "divide and conquer" method. Another possibility is *concretizing* one of the automata to one that only accepts a single string and accepting the loss in test coverage. Finally a list of the dependencies between different symbolic values can be maintained. When the automaton of a string is refined, all symbolic values in that string's list of dependents will be notified so that they can also be refined if necessary. These lists require some overhead, both in storing the dependencies and propagating refinements to multiple automata, but they ensure the most accurate representation of all possible string values at any point in the code. We will discuss how to maintain these lists in Section 4.3.

String predicates are not limited to comparing two strings. Some may have no other inputs besides the string, while others may compare a string with a primitive. The latter can cause symbolic value dependencies between a string and a primitive just like comparing two strings can cause a dependency.

3.2. Operations Returning Primitives

In addition to predicates there are other operations which take strings as inputs. The symbolic semantics of these operations should consider all the possible strings that the input automata can accept. If an operation returns a symbolic

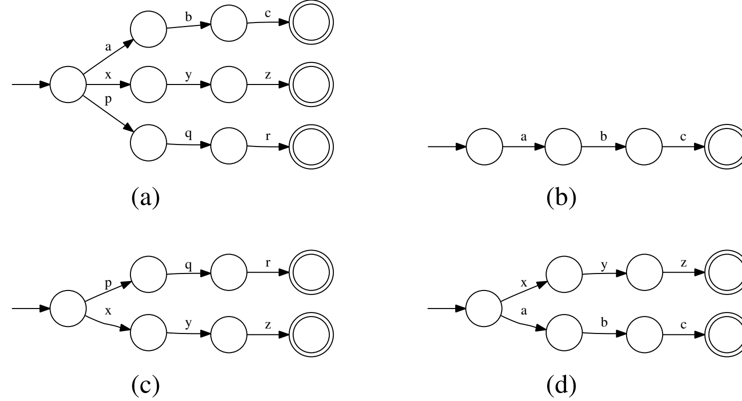

Figure 1. Example symbolic string values (represented as automata) to illustrate symbolic execution over `x.equals(y)`.

value then there is a symbolic value dependency between the input string and the output.

Two operations in Java that both take strings as input and return integer values are `indexOf()` and `length()`. When the symbolic integer value returned by one of these operations is compared with another integer value, we could reduce the number of possibilities for that symbolic value. This could in turn reduce the possible strings that our symbolic string value could be. Consider this example:

```
int l = s.length();
if (l < 4)
  // s must not be longer than three
  //   characters
else
  // s must be four or more
  //   characters long
```

`l` is the length of `s`, so anything we learn about `l` should be reflected in `s`'s automaton. For the path when the `if` branch is taken we know that `l` is less than four, so the automaton for `s` should be refined so it only accepts strings with no more than three characters. When executing the path where the `else` branch is taken, `s` should be refined to only accept strings longer than three characters.

3.3. Operations Returning Strings

There are many operations which can modify or create strings. These include `trim()` and `valueOf()` in Java. These operations can have many inputs including both primitives and strings. Clearly there is a symbolic value dependency between the output string and the inputs. The following example illustrates the necessity of maintaining these relationships.

```
s2 = s1.substring(2);
if (s2.startsWith("abc"))
  if (s1.startsWith("xyz"))
    // this point should be
    //   unreachable
```

The first line of this code should create an automaton which accepts all of the strings accepted by `s1`'s automaton with the first two characters removed. Assuming that `s1` accepts all strings, then the new automaton assigned to `s2` will also accept all strings. When following the path where the `startsWith()` method in the first `if` statement returns true, we will refine the automaton for `s2` to accept all strings which start with “abc”. Additionally we need to refine `s1`'s automaton to accept all strings where the third through fifth characters are “abc”. When we do that, the second `startsWith()` call can only return false because the third character of `s1` is known to be ‘a’. If we did not maintain the dependency between `s1` and `s2`'s automata, we would have executed an impossible path.

4. Implementation

We developed Haderach as a prototype which allows symbolic execution over strings. Haderach is written in Java and extends from the code base of Juzi [13], a relatively simple symbolic executor designed for data structure repair, and from Moller's automaton API [17]. It takes Java programs as input for symbolic execution. Next, we will provide an overview of Juzi (Section 4.1), and discuss the symbolic string class (Section 4.2) and dependencies between symbolic values (Section 4.3).

4.1. Juzi

Juzi uses the Java Programming Assistant (Javassist) [4] and Byte Code Engineering Library (BCEL) [7] to perform bytecode instrumentation, which enables symbolic execution. Primitive types are replaced by custom-designed classes to represent expressions for both symbolic and concrete values. Operations on primitive values are transformed into appropriate method invocations of these symbolic classes. Operations that return boolean values are replaced with ones that internally use a non-deterministic boolean choice when they cannot be deterministically evaluated on-the-fly.

To support non-deterministic choice, Juzi implements a simple backtracking algorithm. The class `Explorer` provides the method `choose()` which takes an integer input `i` and returns a non-deterministic choice between zero and `i`. Such non-deterministic choice operators are an essential feature of software model checkers [9, 11, 20]. The Juzi backtracking algorithm performs stateless depth-first search (i.e., stores no states but remembers the values it uses when making non-deterministic assignments with `choose()`). Non-deterministic code is thus re-executed from the beginning, and during each execution one of the non-deterministic assignments is made differently from that in the previous execution.

Juzi checks satisfiability of path conditions using the CVC Lite [2] automated theorem prover. CVC Lite provides a C++ API for checking validity of formulas over several interpreted theories including linear arithmetic on integers and reals, arrays and uninterpreted functions.

4.2. SymbolicString Class

To add symbolic string values to Juzi's symbolic executor we defined a new Java class: `SymbolicString`. During instrumentation, in addition to replacing primitive types, the `java.lang.String` class is replaced with our `SymbolicString` class.

We use the `dk.brics.automaton` package to implement the automaton described in the Section 3. Each `SymbolicString` object contains an instance of this package's `Automaton` class. The `dk.brics.automaton` package was developed by the authors of [5] to handle the automata in their JSA application. It is a finite-state automaton implementation using the Unicode alphabet for state transitions and supports regular expression operations [17].

For our purposes of solving constraints, the `Automaton` class provides some useful functions. Adding constraints to the automaton is enabled by `intersection`. The old automaton is intersected with a filter to create a refined automaton. This refinement filter is an automaton which accepts all strings allowed by the new constraint. The `isEmpty()` method allows us to test for satisfiability of our symbolic string values enabling us to solve path conditions, which CVC Lite does for symbolic numeric values. Additionally, `getShortestExample()` supplies us with a witness for our path condition, which CVC Lite does not currently support for numeric values. Consequently, we can generate witnesses for paths only containing string constraints.

Eventually we will reproduce in `SymbolicString` all the public methods of the `String` class to support symbolic execution. For now, as a proof of concept, we have implemented a set of key methods illustrated in our test cases, including `startsWith()`, `substring()`, and `trim()`. Our work is not restricted to just the `java.lang.String` class. We can easily address a number of classes implementing the `java.lang.CharSequence` interface. To demonstrate this flexibility, we have developed other classes that thinly wrap our `SymbolicString` class. `SymbolicStringBuffer` and `SymbolicStringBuilder` abstract `StringBuffer` and `StringBuilder`, respectively. We also created `SymbolicPreparedStatement` which abstracts `java.sql.PreparedStatment`. This allows SQL queries to be built and checked for proper grammar.

4.3. Symbolic Value Dependencies

As discussed in Section 3, symbolic value dependencies can dramatically complicate symbolic execution over strings. Concretization of fixed-strings or primitives can alleviate this complexity, but we feel maintaining lists of dependencies allow for a more accurate representation of the possible string values. For our implementation, each `SymbolicString` object contains such a list of dependencies. In addition to keeping a reference to the dependent string, the list tracks the type of relationship. For example after following a path where `s1.startswith(s2)` resolves to `true`, `s1`'s dependency list would contain an entry specifying that it starts with dependent `s2`. `s2`'s dependency list would contain an entry specifying that dependent `s1` starts with it.

When a `SymbolicString` is created from other `SymbolicString` objects, those objects are added to the new `SymbolicString`'s list of dependencies and the new `SymbolicString` is added to their lists. Additionally, if two `SymbolicString` objects are compared by calling a predicate method, these `SymbolicString` objects are added to each other's dependency lists.

These lists of dependencies are used to ensure that established relationships between `SymbolicString` objects remain true. Whenever the automaton of a `SymbolicString` is refined, each of that `SymbolicString`'s dependents is instructed to refine its automaton as well. This causes a

chain reaction of refinements propagating through the dependency hierarchy.

Dependencies between primitives and SymbolicString objects are handled in a similar fashion. The list of dependencies in each SymbolicString can have dependents which are primitives. Refining the SymbolicString's automaton may cause constraints on these primitives to be added to the path condition. Also, each primitive has a list of dependencies. When a constraint is added to the path condition, each of the primitive's dependents will be refined.

5. Experiments

We present experimental results using three case-studies with Haderach. In the first, we symbolically execute an interesting example derived from [5]. This allows us to compare the benefits of our implementation relative to JSA's. Additionally, we run some benchmark programs from [5], showing our ability to perform syntax analysis. Finally, we run Haderach on a well-established Java library, Hibernate, demonstrating Haderach's applicability to a real-world application.

5.1. Tricky

The Java String Analyzer (JSA), a tool developed by Christensen et al. [5], tests for properties of strings by employing a variant of static analysis. JSA generates a finite-state automaton for a regular grammar representing the possible values of string expressions at user-specified hotspots in a Java program. We can examine the string analysis potential of Haderach by comparing its results to JSA'a for a test program.

Tricky is the running example in [5]'s explanation of JSA. This program generates strings of the form "((((((((8*7)*6)*5)+4)+3)+2)+1)+0)". Because of the complexity of its dynamic string construction, it suitably challenges string analysis.

Symbolic executors rely on theorem provers to solve constraints on a path condition. Because these theorem provers have difficulty solving constraints with integer division, we work with a variation of Tricky that maintains a very similar semantic but has no division operations. Listing 2 presents this slightly modified version of Tricky.

When running Haderach on the Tricky program, we begin symbolic execution with foo(). This method takes an integer n as an unknown input, and uses n as the condition for both a loop and a recursive function call. Because n has an infinite number of possibilities, Tricky has an infinite number of execution paths through the loop and recursion. Haderach, like many symbolic executors, cannot handle infinite paths, so we must limit the range of possible values for n.

Listing 2. The Tricky class

```
public class Tricky {
  String bar(int n, int k, String op) {
    if (k == 0)
      return "";
    return op + n + "]"
        + bar(n - 1, k - 1, op) + "␣";
  }

  String foo(int n) {
    StringBuffer b = new StringBuffer();
    if (n * 2 < 2)
      b.append("(");
    for (int i = 0; i < n * 2; i++)
      b.append("(");
    String s = bar(n * 2 - 1, n - 1, "*")
        .trim();
    String t = bar(n * 2 - n, n * 2
        - (n - 1), "+").trim();
    return b.toString() + (n * 2)
        + (s + t).replace(']', ')');
  }
}
```

Figure 2 shows the automaton that Haderach generates for Tricky when n is limited to the values from 1 to 3. For the sake of readability, in this figure <int> represents a sub-automaton which accepts all integer values. With a range for n of 1 to 3, the symbolic execution took 22.5 seconds and explored 27 paths. With a range from 1 to 6, it took 143.5 seconds and explored 69 paths. We ran these tests on a 800 MHz Pentium III with 256MB RAM.

JSA's analysis produces the automaton in Figure 3. We can see a few interesting limitations in this result. First is that JSA does not distinguish between the '*' and '+' characters, even though all strings generated by the program will have no '*'s after the first '+'. With polyvariant analysis [5] claims to be able to fix this problem and refine the result to Figure 4. Another problem lies with the parentheses. The program's output will always have matched pairs of parentheses, but JSA is unable to retain that information in its analysis. This simple issue of maintaining pairs can be crucial in many applications, including the SQL database queries we will discuss later in this section. A more complicated symptom of the same problem is visible in the relationship between the number of '*' and '+' characters. There should always be two more '+'s than '*'s. By assembling the automata as it steps through the code, Haderach is able to maintain this relationship information, as shown in Figure 2.

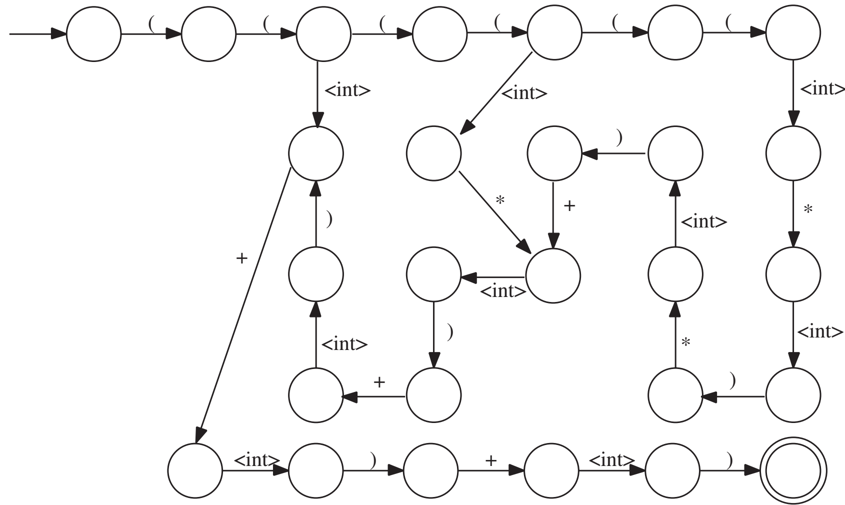

Figure 2. The automaton of Tricky generated by Haderach.

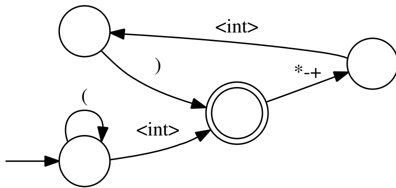

Figure 3. The automaton of Tricky generated by JSA.

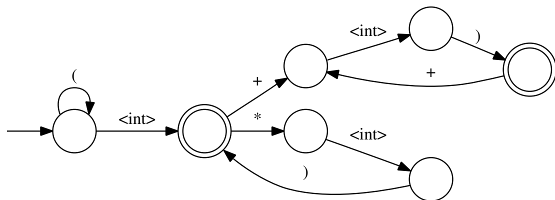

Figure 4. The improved automaton of Tricky generated by JSA.

We can also see in these automata that neither JSA nor Haderach restrict integers in the string to only the values Tricky can possibly generate. Instead both allow for any integer value whenever an integer can appear. For JSA this is a limitation of its method of static analysis, but for Haderach this is a limitation of CVC Lite. By switching to a constraint solver that can provide witnesses to the path condition, Haderach should be able to restrict numbers included in strings to the appropriate values.

Haderach and JSA's strengths and weaknesses complement each other. This balance shows us that neither is inherently superior to the other. Instead they are each useful for different situations.

5.2. Syntax Analysis

In addition to Tricky, the JSA paper [5] contained experimental results from analyzing a series of benchmark programs obtained from the Web. We ran Haderach on a subset of these programs which build and execute SQL queries during runtime. During symbolic execution, when an SQL query is executed, the `SymbolicString` representing that query is saved in a list. After symbolic execution is completed, these `SymbolicString` objects can be compared to an automaton representing the SQL grammar. If the strings accepted by their automata are not a subset of the strings accepted by the SQL grammar then a syntax error could occur in the benchmark program.

Table 1 shows the results of these experiments. We report the number of lines of Java code, the total number of times that hotspots are encountered during symbolic execution, the number of paths explored, the time in millisec-

Example	Lines	Hotspots	Paths	Time	Errors
Decades	26	8	1	2244	0
SelectFromPer	51	1	1	67	0
LoadDriver	78	1	1	686	0
DB2Appl	105	4	2	82	0
AxionExample	162	14	1	927	0
Sample	178	8	1	1094	0

Table 1. Syntax Analysis

onds to symbolically execute the program and record the `SymbolicString` objects at hotspots, and the number of errors encountered. Hotspots are the points in the program when SQL queries are executed. Times were calculated on a 2.33 GHz Core 2 Duo with 2GB RAM running Linux. Comparing the generated automata with the SQL grammar detected no errors. This is consistent with the results achieved by JSA.

5.3. Hibernate

One practical area to apply Haderach is in examining the dynamically generated SQL statements created by database applications. These programs usually create an SQL query based on a variety of parameters including user input. The queries are then sent to a separate database server for execution. Using Haderach to find the possible strings that a program can send to the database, we can check if these statements conform to the SQL grammar. Additionally, we could check them against the database specification or for SQL injections (potentially malicious code inserted in user-controlled portions of the command).

We ran our implementation on code from Hibernate, an open-source database interfacing software [12]. It is commonly used in Java applications to allow developers to freely utilize a database back-end without learning the SQL syntax. The Java class `QuerySelect` shown in Appendix A was extracted from a string rendering helper package `org.hibernate.sql` in Hibernate. It returns an SQL select statement based on the values of certain pre-defined class members, such as table names or parts of the where clause.

Symbolic execution of `toQueryString()` explored 256 paths and took 224.3 seconds on a 800 MHz Pentium III with 256MB RAM. We can perform various checks on the resulting automaton. The most obvious test is to compare it with a test automaton representing the complete SQL grammar [5]. If the possible results are not a subset of the SQL language then there is an error in the program. Additional checks can be made that are more specific to the context of the application. For example, if the database structure is known, then the test automaton could be checked to only allow references to table names which are actually in the database. SQL injection could also be detected with a specialized grammar.

6. Discussion and Conclusions

We present a novel technique that breaks through traditional barriers of symbolic execution by abstracting over higher level data types. Using this methodology, we expand symbolic executions with support for data structures such as strings. Haderach, our implementation based on finite-state automata, provides a new approach to string analysis.

Our work supports symbolic execution over most methods from the `java.lang.string` class and selected methods from the `stringBuilder` and `stringBuffer` classes. In addition, innovative dependency hierarchy tracking and update procedures address issues with the complex relations between symbolic objects. We demonstrate the effectiveness and the scalability of our implementation through experiments. Our test subjects include benchmark examples obtained from academia and well-known application used in the industry. The practical applications of our work are not limited to the verification of SQL grammar, but any dynamically generated language, such as HTML output from Java Servlets.

In our future work, we plan to take steps towards integrating an algebraic solver to address limitations with the current theorem prover. This enhancement will complete our support for symbolic execution of string methods involving primitive arguments. In addition, we plan to port Haderach into the Java PathFinder model checker [10] to set the ground-work for extending symbolic execution to an even wider range of useful data structures.

Acknowledgments

This work was funded in part by NSF ITR-SoD award #0438967.

References

[1] T. Andrews, S. Qadeer, S. Rajamani, J. Rehof, and Y. Xie. Zing: A model checker for concurrent software. In *16th International Conference on Computer Aided Verification (CAV)*, Boston, MA, July 2004.

[2] C. Barrett and S. Berezin. CVC Lite: A new implementation of the cooperating validity checker. In *Proceedings of the 16th International Conference On Computer Aided Verification*, Boston, MA, July 2004.

[3] W. R. Bush, J. D. Pincus, and D. J. Sielaff. A static analyzer for finding dynamic programming errors. *Software—Practice and Experience*, 30(7), 2000.

[4] S. Chiba. Javassist—a reflection-based programming wizard for Java. In *Proceedings of the ACM OOPSLA'98 Workshop on Reflective Programming in C++ and Java*, Oct. 1998.

[5] A. Christensen, A. Møller, and M. Schwartzbach. Precise analysis of string expressions. In *Proc. 10th International Static Analysis Symposium (SAS'03)*, June 2004.

[6] C. Csallner and Y. Smaragdakis. Check 'n' crash: Combining static checking and testing. In *Proc. 27th International Conference on Software Engineering (ICSE)*, St. Louis, MO, May 2005.

[7] M. Dahm. Byte code engineering library. http://bcel.sourceforge.net/.

[8] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *Proc. ACM SIGPLAN 2002 Conference on Programming language design and implementation*, 2002.

[9] P. Godefroid. Model checking for programming languages using VeriSoft. In *Proc. 24th Annual ACM Symposium on the Principles of Programming Languages (POPL)*, Paris, France, Jan. 1997.

[10] K. Havelund and T. Pressburger. Model checking Java programs using Java PathFinder. *International Journal on Software Tools for Technology Transfer*, 1999.

[11] G. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5), May 1997.

[12] JBoss. Hibernate core. http://www.hibernate.org/.

[13] S. Khurshid, I. García, and Y. L. Suen. Repairing structurally complex data. In *12th SPIN Workshop on Model Checking of Software*, San Francisco, CA, Aug. 2005.

[14] S. Khurshid, C. Pasareanu, and W. Visser. Generalized symbolic execution for model checking and testing. In *Proc. 9th Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, Warsaw, Poland, April 2003.

[15] S. Khurshid and Y. L. Suen. Generalizing symbolic execution to library classes. In *6th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, Lisbon, Portugal, Sept. 2005.

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

[17] A. Møller. Brics automaton library. http://www.brics.dk/automaton.

[18] C. S. Pasareanu and W. Visser. Verification of java programs using symbolic execution and invariant generation. In *Proc. 11th International SPIN Workshop on Model Checking of Software*, Barcelona, Spain, Apr. 2004.

[19] M. Veanes, C. Campbell, W. Schulte, P. Kohli, N. Tillmann, and W. Grieskamp. On-the-fly testing of reactive systems. (Submitted for publication.).

[20] W. Visser, K. Havelund, G. Brat, and S. Park. Model checking programs. In *Proc. 15th Conference on Automated Software Engineering (ASE)*, Grenoble, France, 2000.

[21] T. Xie, D. Marinov, W. Schulte, and D. Notkin. Symstra: A framework for generating object-oriented unit tests using symbolic execution. In *Proc. 11th Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, Edinburgh, UK, Apr. 2005.

A. Hibernate QuerySelect

```
public class QuerySelect {
  private StringBuffer select = new StringBuffer();
  private StringBuffer where = new StringBuffer();
  private StringBuffer groupBy = new StringBuffer();
  private StringBuffer orderBy = new StringBuffer();
  private StringBuffer having = new StringBuffer();
  private String joins_toFromFragmentString;
  private String joins_toWhereFragmentString;
  private String comment;
  private boolean distinct = false;

  ...

  public String toQueryString() {
    StringBuffer buf = new StringBuffer(50);
    if (comment != null)
      buf.append("/*␣").append(comment).append("␣*/␣");
    buf.append("select␣");
    if (distinct)
      buf.append("distinct␣");
    String from = "";
    if (from.startsWith(",")) {
      from = from.substring(1);
    } else if (from.startsWith("␣inner␣join")) {
      from = from.substring(11);
    }

    buf.append(select.toString()).append("␣from").append(from);

    String outerJoinsAfterWhere = "";
    String whereConditions = where.toString().trim();
    boolean hasOuterJoinsAfterWhere = outerJoinsAfterWhere.length() > 0;
    boolean hasWhereConditions = whereConditions.length() > 0;
    if (hasOuterJoinsAfterWhere || hasWhereConditions) {
      buf.append("␣where␣");
      if (hasOuterJoinsAfterWhere) {
        buf.append(outerJoinsAfterWhere.substring(4));
      }
      if (hasWhereConditions) {
        if (hasOuterJoinsAfterWhere) {
          buf.append("␣and␣(");
        }
        buf.append(whereConditions);
        if (hasOuterJoinsAfterWhere) {
          buf.append(")");
        }
      }
    }

    if (groupBy.length() > 0)
      buf.append("␣group␣by␣").append(groupBy.toString());
    if (having.length() > 0)
      buf.append("␣having␣").append(having.toString());
    if (orderBy.length() > 0)
      buf.append("␣order␣by␣").append(orderBy.toString());

    return buf.toString();
  }

  ...

}
```