String Analysis for Java and Android Applications

Ding Li, Yingjun Lyu, Mian Wan, William G. J. Halfond
University of Southern California
Los Angeles, California, USA
{dingli, yingjunl, mianwan, halfond}@usc.edu

ABSTRACT

String analysis is critical for many verification techniques. However, accurately modeling string variables is a challenging problem. Current approaches are generally customized for certain problem domains or have critical limitations in handling loops, providing context-sensitive inter-procedural analysis, and performing efficient analysis on complicated apps. To address these limitations, we propose a general framework, Violist, for string analysis that allows researchers to more flexibly choose how they will address each of these challenges by separating the representation and interpretation of string operations. In our evaluation, we show that our approach can achieve high accuracy on both Java and Android apps in a reasonable amount of time. We also compared our approach with a popular and widely used string analyzer and found that our approach has higher precision and shorter execution time while maintaining the same level of recall.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging

General Terms

Algorithms

Keywords

String analysis, mobile apps

1. INTRODUCTION

Strings play an important role in modern software. The server side of modern web applications may use strings to output HTML content and make SQL queries to databases, while the client side may process string-based user input and send it to the server-side via various string-based protocols. Mobile apps may generate string-based HTTP requests to cloud-based servers, and even large scale systems, such as Distributed Event-Based (DEB) systems, may pass messages whose attributes are represented as strings. By analyzing these strings and determining their possible values, engineers can better understand an application's potential runtime behaviors. For example, string analysis has been used to detect SQL injection attacks [19, 38, 39, 14], prevent Cross-Site Scripting attacks [24, 40], detect errors in HTML output [29, 21], optimize energy consumption of mobile web apps [26], improve penetration testing [22], and understand the runtime behavior of DEB systems [15, 30]. Inaccurate string analysis can reduce the effectiveness of these approaches. For example, an imprecise string analysis can lead to the identification of less energy-efficient color schemes (i.e., [26]); and an unsafe string analysis could miss possible vulnerabilities (e.g., [40]).

Despite their widespread use and importance, accurately analyzing strings remains a significant challenge due to several issues: (1) how to analyze string values generated in (possibly nested) loops that may include loop carried data dependencies, (2) context-sensitivity for strings manipulated and composed inter-procedurally, (3) flexibility to attach different semantics to string operations, and (4) scalability to handle long strings or strings manipulated using a complex series of operations. Existing techniques that use string analysis have mainly side-stepped these challenges by cleverly leveraging aspects of their problem domain to simplify the required string analysis. The result is these key challenges for the analysis of strings have not been adequately addressed.

Existing work related to string analysis can be broadly described as either performing string analysis in support of another software engineering goal or directly providing a string analysis technique. Approaches that fall into the first category include those for approximating HTML output (e.g., [29, 21]), computing possible SQL queries (e.g., [19, 38, 39, 14]), and identifying messages passed in DEB systems [15]. These approaches have often not needed to address key challenges in string analysis due to aspects of their problem domains. For example, to optimize display energy [26], it was sufficient to assume that loops were unraveled once. The result of leveraging these domain-specific insights is that the developed techniques are not generalizable, since assumptions that work for one problem domain may not be appropriate for others. This has implications for the research community as each group that wants to develop techniques that need string analysis must essentially start from scratch.

More generalizable techniques have also been proposed (e.g., [9, 41]). However, these have trade-offs in how they handle the four challenges, which makes them less broadly applicable. For example, JSA [9] uses global alias analysis to model inter-procedural manipulation of strings, which, as we show in Section 4, leads to scalability problems when analyzing Android applications that include extensive invocations of framework APIs. Yu and colleagues [41] proposed a Finite State Automata (FSA) based *Widen* operation, which can partially solve the first challenge, but is not able to handle nested loops. Symbolic execution based techniques can more precisely address challenges 1, 2, and 3. However, these techniques may not scale for large programs and may make simplifying assumptions about the strings under analysis.

The goal of the work presented in this paper is to present a general framework for string analysis that allows researchers to more flexibly choose how they will address each of the four challenges. Our key insight into how to do this is to separate the *representation* and *interpretation* of string operations. Our string analysis framework, Violist, defines an Intermediate Representation (IR) that faithfully represents the string operations performed on an application's string variables. Violist's IR is designed to accurately capture complex data-flow dependencies in loops and context-sensitive call site information. Violist also allows for the straightforward integration of IR interpreters that can implement the user's necessary interpretation of the string construction semantics. For example, it is straightforward to write an interpreter that will unravel loops once, n times, or approximate an upper bound for infinite unraveling. Finally, Violist can easily scale up and analyze large programs.

To evaluate the usefulness and effectiveness of our framework, we carried out an extensive empirical evaluation. For the evaluation, we implemented two different IR interpreters and used these to compare the accuracy and scalability of Violist against the popular Java String Analyzer (JSA) [9]. For this evaluation, we used a mixture of publicly available benchmarks, systematically created test cases to mimic different data and control flows, and real-world Java and Android applications. Our results showed that Violist is able to generate results that, on average, are more precise than JSA while maintaining the same level of recall. Furthermore, Violist was able to generate these results for a wider range of applications and to do this faster than JSA.

2. MOTIVATION

In this section, we provide a motivating example (Program 1) to illustrate the four challenges mentioned in Section 1.

2.1 Loops

Consider a string analysis that targets the variable c at line 19 (c_{19}). This variable is redefined in each iteration of the loop at lines 17–21. Although the upper bound on the loop can be trivially identified via inspection, in general, it is challenging for a static string analysis to accurately account for the loop's upper bound. Therefore, many techniques (e.g., [9]) simply assume the loop will be executed an infinite number of times or unrolled only once (e.g., [29, 26]). Even techniques that do model loops are generally unable to model the relationship of nested loops, an example of which is shown at lines 27–35. Nested loops are more challenging to analyze, because it is necessary to model the relationship of the strings in the inner and outer loops (e.g., d and e).

```
 1 class Example {
 2   public static String addA(String v)
 3   {
 4     return v+"A";
 5   }
 6   public static String replaceA(String v)
 7   {
 8     return v.replaceAll("A","B");
 9   }
10   public static void main(String[] args) {
11     String a="a";
12     String b=addA(a);
13     b=addA("b");
14     System.out.println(b);
15     String c="A";
16     String f="";
17     for(int i=0;i<3;i++)
18     {
19       c=c+"A";
20       f=f+c;
21     }
22     c=replaceA(c);
23     c=replaceA("AAAA");
24     System.out.println(c);
25     String d="";
26     String e="";
27     for(int i=0;i<3;i++)
28     {
29       d=d+e;
30       System.out.println(d);
31       for(int j=0;j<1;j++)
32       {
33         e=e+"b";
34       }
35     }
36   }
37 }
```

Program 1: Example program

Techniques based on the *Widen* approach for approximating loops ([41, 8, 5]) would not be able to handle this scenario. Many flow-based techniques would simply model this as one large loop, which is safe, but reduces precision.

2.2 Context-Sensitivity

Next, consider a string analysis that includes string manipulations that are carried out inter-procedurally. An example of this is in method `main` at lines 12 and 13, which calls the method `addA`. Many string analyses handle this sort of invocations without any call site context sensitivity (e.g., [41, 9]). This means that when analyzing the values at lines 12 and 13, these analyses will assume that the return value can be based on the arguments provided at any call sites to `addA`. Using the example, this means that the value of b at lines 12 *and* 13 would be approximated as {"aA", "bA"} instead of "aA" for line 12 and "bA" for line 13. As with handling of loops, this approach is safe, but loses precision as extra possible string values would be returned as a possible value of the variable b. The reason this occurs is that the analyses use a global data flow analysis, which is, in turn, based on the call graph of the application. This representation of the inter-procedural control-flow results in a lack of context-sensitivity. Techniques that use symbolic execution do not face this problem, but may require SMT problem solving and need to analyze all possible paths of a program, which can make it difficult to scale the techniques for large programs.

2.3 Flexible Semantics

A general limitation of many string based approaches is that they are tightly tied to one specific method of interpreting string values. For example, JSA [9] approximates loops as having an infinite upper bound, both D-model [29] and the string analysis underpinning Nyx [26] assume loops are unraveled once. Beyond that many string analyses are highly customized to make approximations in ways that are appropriate for their problem domains, but that limit their general applicability. For researchers and software engineers who are attempting to leverage string analysis, it is generally necessary to develop their own string analysis to be used for their project. This can be a significant barrier to entry and to the success of the project.

2.4 Scalability

Lastly, symbolic execution techniques could be used to address the context-sensitive problem. However, these techniques may not scale easily for large programs. To improve scalability, symbolic execution techniques may assume strings are bounded in length (e.g., no longer than 8 characters), which limits the techniques' general applicability, or model strings as sequences of characters, which cannot adequately represent the semantics of certain string operations, such as `replaceAll`.

3. APPROACH

Our approach provides a general framework for string analysis. This framework allows for the development of customized string analyses that vary in terms of recall and precision in how they handle loops, context sensitivity, and string operation semantics. To build this framework, we designed an approach that separates the representation of the string operations from their interpretation. This separation allows all string analyses to share a common underlying string model, yet attaches different semantics to the modeled instructions. Our approach can be defined as having two general phases. In the first, the approach builds a model of the string operations in the Program Under Test (PUT), and then in the second, applies a custom interpreter to the model that implements the desired string operation semantics.

To model the string operations, we define an IR that captures the data flow dependencies between string variables and string operations. The IR can be computed for any code that can be represented in a Static Single Assignment (SSA) form, for which translations exist for most mainstream language (e.g., Java, Dalvik, and PHP). In addition, the IR also includes operations that allow it to represent strings defined externally to the block of code (e.g., method parameters) and data dependencies caused by loops. We define the details of the IR in more depth in Section 3.1.

In the first phase, our approach analyzes the PUT and computes an IR-based summary for each of its methods. Within each method, the approach uses a region-based analysis to identify code enclosed by (possibly nested) loops and then specially models the data dependencies caused by loops. In the second phase, our approach translates the IR-based summaries into a string representation. To do this, our framework allows a user to supply an interpreter of their choice that implements the desired semantics with respect to string operations, loops, and context sensitivity. As part of the evaluation, we implemented two such interpreters, one that models the strings as FSAs (as is done in JSA [9]) and the second that carries out an n-bounded loop unraveling for all loops in the PUT. The interpreters also allow the users to leverage additional analyses, such as alias analysis, that can more precisely identify loop upper bounds, include domain-specific knowledge, or resolve other constraints.

3.1 The Intermediate Representation

We define an IR that represents the control and data flow relationships among the string variables and string operations in a program. The goal of the IR is to precisely model these relationships while deferring any sort of interpretation or approximation of these relationships until the second phase. Our IR specifically targets control and data flow relationships that have traditionally complicated the modeling and interpretation of strings. Namely, strings defined external to the analysis scope, strings constructed within (possibly nested) loops, and inter-procedural string manipulation.

Our IR is structured as a tree with the leaf nodes defining either string constants or placeholders for unknown variables. The internal nodes in the IR tree are string expressions that represent the values of string variables in the PUT. For explanatory purposes in the paper, we will represent expressions in the IR tree as $(op\ a_1, a_2, ...)$, where op is the operation of the expression and a_i represents the various arguments to op. We write the definition of a variable in the form of v_l, where v is the source-code based name of the variable and l is the line number of the definition. To illustrate, line 13 of Program 1 is represented as $b_{13} = (addA\ \text{“}b\text{”})$.

In cases where a variable is defined outside of the scope of the analysis (e.g., method parameters), our approach leaves placeholders in the IR. A placeholder variable is denoted by the subscript notation of “X_n” where X is a fixed symbol and n is a number, unique within the analysis scope, identifying the external variable. For example, consider line 4 of Program 1, the IR for this line is $(+\ v_{X4}\ \text{“}A\text{”})$. These placeholders are preserved until they can be resolved. For example, in this case the IR of line 4 becomes the method summary for $addA$. Whenever a call site for $addA$ is encountered, say at line 13, the argument can be provided for the placeholders. Placeholders also allow additional analysis to be employed by the interpreters to resolve strings that may originate from files or database queries.

Our approach defines several IR operators to model the effects of loops. The first of these is τ, which is used to represent string variables whose values are partly defined via a loop carried data dependency. The form of this operator is $(\tau^{T}_{v.r}\ (exp))$. Here v is the name of the variable defined in the loop; r is the ID of the loop (our approach numbers each loop region, as explained in Section 3.3); T represents the number of the loop iterations, with $T = 0$ denoting the initial value of a variable in a loop and $T = 1$ denoting the value of a variable after one iteration; and exp is the IR expression that contains v in the loop. Note that exp may contain additional τ expressions, which enables nested loops to be easily modeled. Within exp, the defined variable v is represented using the symbol $\sigma^{T'}_{v.r}$.

To illustrate, consider the variable e_{33} in Program 1. Equation (1) shows the value of e_{33} for the k^{th} iteration.

$$(\tau^{k}_{e_{33}.R2}\ (+\ \sigma^{k-1}_{e_{33}.R2}\ \text{“}b\text{”})) \qquad (1)$$

This equation shows that the value of e at line 33 is equal to the concatenation of its value from the previous iteration and the constant string "b". Here, the identifier “$R2$” refers to the loop region ID, which we will explain in more detail in Section 3.3.

The final loop related operator is ϕ. The ϕ expression has the same format as the τ expression, but denotes a dependency between self-referring variables in the same loop instead of a nested loop dependency. Here, we define “self-referring variable" as a variable that defines itself, at least in part, via a loop carried dependency. The ϕ expressions are always sub-expressions of the τ expressions. When a ϕ expression is used, it represents that the variable of the τ expression depends on the variable of the ϕ expression and both of them are from the same loop.

To make this clear, we take the example at line 19 and line 20 in Program 1. In our example, both c_{19} and f_{20} are self-referring variables from the same loop and c_{19} is used to define f_{20}. The IR of f_{20} is shown in Equation (2), where the underlined portion refers to the value of c_{19}, which shows how the value of f_{20} depends on the value of c_{19}. In this expression, the superscript of 0 means the value of the variable in the current iteration and $-n$ means the value n iterations prior.

$$f_{20} = (\tau^{0}_{f_{20}.R3}\ (+\ \sigma^{-1}_{f_{20}.R3}\ (+\ \underline{(\phi^{-1}_{c_{19}.R3}(+\ \sigma^{-2}_{c_{19}.R3}\ \text{“}A\text{”}))}\ \text{“}A\text{”}))) \tag{2}$$

We introduce the ϕ symbol because the dependency between self-referring variables in the same loop is different from the dependency due to nested loops. We compare the value of f_{20} in Equation (2) to the IR of d_{29} in Equation (3). In the IR of d_{29}, the underlined portion refers to the value of e_{33}, which is generated in a loop different from that of d_{29}. On the contrary, in the IR of f_{20}, the underlined portion refers to c_{19}, which is a variable from the same loop of f_{20}.

$$d_{29} = (\tau^{0}_{d_{29}.R1}\ (+\ \sigma^{-1}_{d_{29}.R1}\ \underline{(\tau^{-1}_{e_{33}.R2}\ (+\ \sigma^{-2}_{e_{33}.R2}\ \text{“}e\text{”}))})) \tag{3}$$

3.2 Interpretation of the Intermediate Representation

The interpretation step converts the IR of string variables into a representation of strings. Our framework allows users to provide an interpreter that translates the IR as needed for different analysis problem. To do this, a user would need to implement two things: (1) the string model for each string operation (e.g., `append` and `trim`); And (2) a $Widen$ and $Converge$ function for the τ expressions. Here, $Widen$ is used to generalize the τ expression on each iteration to make the string values converge quicker. For example, one instance of $Widen$ proposed by Yu and colleagues [41] can generalize string set {"a", "aa", "aaa"} to the regular expression $a+$. $Converge$ is used to judge when the approach should terminate iterating over the τ expression.

The general process for IR interpretation is as follows. First, our approach uses the target string models to represent all leaf nodes in the IR that are string constants and external inputs. Second, our approach calculates the value of each non-leaf node using a postorder traversal of the IR tree. During this calculation, our approach uses the specified string operation semantics to calculate the value of each IR operation. Third, when the approach encounters a τ expression, it iterates over the expression to calculate the resulting string value. We will describe this process in more detail in the next paragraph. After each iteration, our approach calls the $Widen$ expression to generalize the string model created in the current iteration, and then the $Converge$ method to check if it should stop iterating over the τ expression.

Our approach may iterate several times over each τ expression, depending on the $Widen$ and $Converge$ operations. In each iteration while interpreting the τ expression ($\tau^{T}_{v.r}$ (exp)), where exp contains at least one $\sigma^{k}_{v.r}$, our approach first increases the counter T for the τ expression and all $\sigma_{v.r}$. After this, our approach sets the value of $\sigma^{k}_{v.r}$ as the value of ($\tau^{k}_{v.r}$ (exp)) generated in the kth iteration. Then, it calculates the value of exp, calls the $Widen$ operation to calculate the result of current iteration. This value will be recorded as the value of σ in future iterations.

During each iteration, if our approach encounters a nested τ expression, it will iterate over this expression until the value converges using the processes we described in the prior paragraph. If it encounters a ϕ expression, the approach updates the counters for ϕ and its σ, and calculates the value with a process similar to that of interpreting the τ expression. The only difference is that our approach does not iterate over the ϕ operation several times. It only calculates the value of ϕ once without calling the $Widen$ and $Converge$. We do this because ϕ represents another variable in the same loop so its value can only be calculated once in each iteration. The resulting values of the ϕ expression will also be recorded as the values of future corresponding σ operators.

3.3 Getting the Intermediate Representation

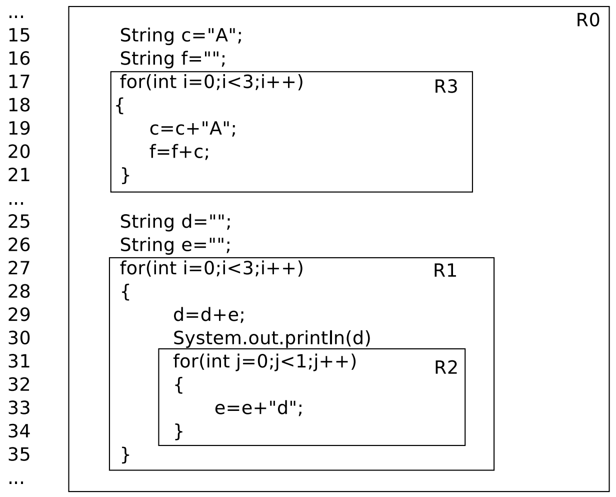
In this section, we describe how our approach generates the IR for a given PUT. As inputs, our approach requires the Call Graph (CG) of the PUT and the SSA representation of each method in the PUT. In general, most mainstream languages (e.g., Java, Dalvik, and PHP) have analyses available that can provide both of these elements. Given these inputs, our approach analyzes each method in reverse topological order with respect to the PUT's CG. For each method, the approach then identifies the Region Tree (RT) and builds the IR for each region in this tree in a bottom up fashion. After the approach finishes analyzing a method, the resulting IR serves as its summary and is reused whenever another method calls the summarized method.

3.3.1 Intra-procedural Analysis

The approach begins with an intra-procedural analysis to calculate the IR based summary for each method in the PUT. The first step of this analysis is to identify the bodies of loops and their relationship to other loops. This information is needed in order to use the loop modeling operands defined by the IR. To identify loops, our approach uses a standard analysis to identify regions in the method's Control-Flow Graph (CFG) [4].

The regions of a method can be represented as a RT in which nested regions are shown as children of their parent regions and the root node of the tree is the method body. Figure 1 shows the RT for an excerpt of the code of Program 1. In this figure, $R0$ represents the method body of `main` and $R1$–$R3$ are loops in the method.

After the approach builds the RT, the next step is to generate the IR for each region. The approach analyzes the RT using a post-order traversal (i.e., starting with the leaf nodes) so that the results of analyzing nested regions can be incorporated into the analysis of the parent regions. In general, there are two types of regions that the approach analyzes. The first type is Method Body Regions (MBRs), which represent the main body of a method (e.g., $R0$), and the second type is Loop Body Regions (LBRs), which are the regions that represent the bodies of loops (e.g., $R1--R3$). We now explain how the approach analyzes each of these region types.



```
...
15      String c="A";
16      String f="";
17      for(int i=0;i<3;i++)
18      {
19         c=c+"A";
20         f=f+c;
21      }
...
25      String d="";
26      String e="";
27      for(int i=0;i<3;i++)
28      {
29            d=d+e;
30            System.out.println(d)
31            for(int j=0;j<1;j++)
32            {
33                 e=e+"d";
34            }
35      }
...
```

Figure 1: The Region Tree for Program 1

Algorithm 1 Getting the intra-procedural intermediate representation

```
Input: A region $R$
Output: The IR of string variables in $R$
 1: $External \leftarrow$ all External Variables
 2: for all Instruction $S$ in $R$ in topological order do
 3:     if $S$ defines a variable $v$ then
 4:         Represent S in the form of the IR as $v = (op\ a_1\ a_2\ ...)$
 5:         for all $a_i \in Par(S)$ do
 6:             if $a_i$ has been solved and $a_i \notin External$ then
 7:                 Replace $a_i$ with its IR
 8:             end if
 9:         end for
10:     end if
11: end for
```

3.3.2 General Region Processing

Regardless of their types, all regions are first processed by Algorithm 1. The input to Algorithm 1 is the SSA representation of the code in a region R. The output of Algorithm 1 is a basic IR transformation of R that does not consider the effect of loops.

The first step of Algorithm 1 is to identify variables whose definitions occur outside of the region (line 1). These variables can be identified in a straightforward way, since the code is in SSA form, by checking if they are defined in the current region. All other variables are considered to be internal variables. The approach will leave placeholders for external variables until their definitions are located in a containing region. An example of an external variable is the use of c on the righthand side of line 19 in $R3$.

After finding all external variables, the approach then iterates over each instruction in R in topological order with respect to R's CFG. For each instruction i in which the left hand side defines an internal variable v, the approach converts i into an IR operand of the form $(op\ a_1\ a_2\ ...)$ as shown in Section 3.1. Then the approach iterates over all arguments for *op*, which is $Par(S)$. If any argument a_i has previously been defined on the lefthand side of an instruction in the region, then the righthand side of that definition is substituted for the a_i. This process is continued until there are no more such previously defined internal variables present in the IR form of i. Then the approach repeats this for each of the remaining $i \in R$. Note that since the effects of loops are not considered at this time and the instructions are processed in topological order, the transformation converges on a fixed point after one iteration over the instructions in R.

Example: To illustrate Algorithm 1, first consider line 19 of R3 in Figure 1. The IR for this line is $(+\ c_{19}\ \text{“}A\text{”})$. Note that due to the SSA transformation, $(+\ c_{15}\ \text{“}A\text{”})$ could also be a definition here. However, we omit these SSA based definitions from the explanation for simplicity, since our code example is not shown in SSA form. Following with this, the IR for line 20 in R3, would be $(+\ f_{20}\ (+\ c_{19}\ \text{“}A\text{”}))$.

3.3.3 Processing Loop Regions

The next step is to analyze all LBRs and more precisely model the effects of loops in the region. This is performed by the algorithm shown as Algorithm 2. The input to Algorithm 2 is the IR of an LBR that has already been processed by Algorithm 1. The output is an IR representation of the LBR that has been adjusted to model the effects of loops. The IR of an LBR can be incorporated into its parent loop or a MBR.

Algorithm 2 can be thought of as having three general phases. The first of these (lines 1–10) rewrites all string variables within the LBR so that they have counters attached to them. These counters are then used in the second phase to identify dependencies between loop iterations. The second phase (lines 11–39) iterates over the instructions in the LBR and replaces string variables defined in the loop with values updated for each iteration. Loop-carried data dependencies are identified and replaced with the τ and σ operators so that the definitions of the string variables can converge on a fixed point. Finally, in the third phase (lines 40–48), the analysis replaces certain τ operators with the ϕ operators. We now explain these three phases in more detail below.

In the first phase (lines 1–10), our approach assigns and initializes a counter to all region-internal string variables. The analysis iterates over each instruction $i \in$ LBR and if it is of the form $v = (op\ a_1\ a_2\ ...\ a_m\ ...\ a_n)$ (i.e., a definition) and v is an internal variable then the analysis performs the following transformations. The variable v is rewritten to be v^0 and each of the arguments a_k is rewritten as a_k^{-1}. The intuition behind this transformation is that the superscripts show that the variable on the lefthand side is defined by the values of the righthand side variables that come from the previous (i.e.,-1) iteration. Note that since Algorithm 1 has already propagated all values forward, all righthand side variables were defined in the previous iteration or, in the base case, external to the LBR.

To illustrate the first phase we will build on the example from Section 3.3.2. Here for line 20 of Program 1, the IR is $f_{20} = (+\ f_{20}\ (+\ c_{19}\ \text{“}A\text{”}))$. The transformed version of this IR with counters inserted and initialized would be $f_{20}^0 = (+\ f_{20}^{-1}\ (+\ c_{19}^{-1}\ \text{“}A\text{”}))$. Note that, after the transformation done by Algorithm 1, the f_{20} on the righthand side refers to the value of f from the previous loop's iteration (or in the base case, the value provided at line 16.)

In the second phase (lines 11–39), the approach iterates over each instruction, propagating values and updating counters until the IR of the instructions converges on a fixed point. The approach begins this phase by iterating over each instruction that is of the form $v^0 = (op\ a_1\ a_2\ ...\ a_n)$ (i.e., a definition). At lines 15–19, the approach adds all variables on the righthand side that are region-internal variables to a worklist Q. Then at lines 20–36, the approach will iterate over all variables in the worklist, replacing them with updated values and introduce τ and σ notions so that variables defined in the loop will converge to a fixed point. In the loop beginning at line 20, the approach dequeues the first argument in the worklist and determines if it refers to v (i.e., if it is a self reference.) If it is, then the approach replaces the self reference with the σ notation and puts a τ notation around the expression, as described in Section 3.1. If the argument is not a self reference, then the approach replaces it with the most up-to-date IR representation and decreases all counters in the replaced expressions by the counter value for the replaced argument. (This maintains the correspondence between loop iterations.) All arguments present in the replaced argument that are self-references are then also added to the worklist. This process repeats until there are no more arguments to be resolved in the worklist.

To illustrate the second phase, consider again the example from above. For line 19 in R3, Our approach will add the τ expression and change c_{19} to $\sigma_{c_{19}.R3}$. Therefore the IR for line 19 is as shown in Equation (4).

$$c_{19} = (\tau^0_{c_{19}.R3}\ (+\ \sigma^{-1}_{c_{19}.R3}\ \text{“}A\text{”})) \tag{4}$$

Similarly, for line 20 in R3, we replace the reference c^{-1}_{19} with Equation (4) and the IR for line 20 will be as shown in Equation (5). Here, the underlined portion is the part that came from Equation (4).

$$f^0_{20} = (\tau^0_{f_{20}.R3}\ (+\ \sigma^{-1}_{f_{20}.R3}\ (+\ \underline{(\tau^{-1}_{c_{19}.R3}(+\ \sigma^{-2}_{c_{19}.R3}\ \text{“}A\text{”}))}\ \text{“}A\text{”}))) \tag{5}$$

The third phase of the approach (lines 40–48) introduces the ϕ operator that denotes the data-dependency between variables in the same loop region. This enables the interpreters to distinguish region-internal variables from the same loop and nested loops. To do this, our approach iterates over all τ expressions and checks their region IDs. If it finds that there is a τ expression $\tau_{a.R}$ embedded by another τ expressions $\tau_{b.R}$ with the same region ID, the approach changes $\tau_{a.R}$ to $\phi_{a.R}$. This is because if two τ expressions have the same region ID, they are generated in the same loop.

To illustrate, consider again line 20 in R3. After the third phase, its IR will be as shown in Equation (2). Here, the symbol $\phi^{-1}_{c_{19}.R3}$ is the changed portion.

3.3.4 Inter-procedural Analysis

Once the IR has been calculated for each method's MBR, then analysis of the method is complete. The IR for the MBR is used as the method's summary to enable inter-procedural analysis. When an invocation to the summarized method is encountered during the analysis of Algorithm 1, the IR of the summarized method's return variable is inserted into the current method body and placeholder variables in the IR are replaced with the arguments provided at the invocation call site. Note that methods are processed in reverse topological order with respect to the PUT's CG so that a method's summary is computed before that of any calling method. In the case that there are Strongly Connected Components (SCC) in the CG, our approach builds a global CFG for the SCC and then uses the intra-procedural analysis to generate the IR for this connected method.

Algorithm 2 Solving the loop region

```
Input: A loop region $R$
Output: The IR of string variables in $R$
 1: for all Instruction $S$ in $R$ in topological order do
 2:     if $S$ defines a variable $v$ then
 3:         $v \rightarrow v^0$
 4:         for all $a_i \in Par(v)$ do
 5:             if $a_i \notin External$ then
 6:                 $v \rightarrow a_i^{-1}$
 7:             end if
 8:         end for
 9:     end if
10: end for
11: while the IR for all variables are not converged do
12:     for all Instruction $S$ in $R$ in topological order do
13:         if $S$ defines a variable $v$ then
14:             $Q \leftarrow EmptyQueue$
15:             for all $a_i^{-k} \in Par(v)$ do
16:                 if $a_i \notin External$ then
17:                     Add $a_i^{-k}$ to the tail of $Q$
18:                 end if
19:             end for
20:             while $Q$ is not empty do
21:                 $a_i \leftarrow Q.poll()$
22:                 if $a_i$ equals to $v$ then
23:                     Replace $a_i^{-k}$ with $\sigma_{v.rid}^{-k}$
24:                     if There is no $\tau_{v.rid}^0$ in the IR of $v$ then
25:                         Put $\tau_{v.rid}^0$ around the IR of $v$
26:                     end if
27:                 else
28:                     Replace $a_i^{-k}$ with its IR
29:                     Decrease all counters in $a_i$ by k
30:                     for all $b_i^{-j} \in Par(\,a_i^{-k})$ do
31:                         if $b_i^{-j}$ refers to $v$ then
32:                             Add $b_i^{-j}$ to the tail of $Q$
33:                         end if
34:                     end for
35:                 end if
36:             end while
37:         end if
38:     end for
39: end while
40: for all variable $v$ in the Region do
41:     if $v$ contains a $\tau$ expression $\tau_{v.RID}$ then
42:         for all $\tau_{x.rid}$ contained by $\tau_{v.RID}$ do
43:             if $rid$ equals to $RID$ then
44:                 $\tau_{x.rid} \rightarrow \phi_{x.rid}$
45:             end if
46:         end for
47:     end if
48: end for
```

3.4 Illustrative Example of the Analysis

We walk through our algorithm using Program 1 as an example. Our approach first builds the summary for each region in a bottom-up order. The approach first analyzes the leaf regions, R3 and R2. R3 is a LBR that defines two variables, c_{19} and f_{20}. As we described earlier in Section 3.3.2, we initially represent their values as $c_{19} = (+\ c_{19}\ \text{“}A\text{”})$ and $f_{20} = (+\ f_{20}\ c_{19})$. Then the approach replaces the symbol c_{19} in f_{20} with its definition so that we have $f_{20} = (+\ f_{20}\ (+\ c_{19}\ \text{“}A\text{”}))$. After this, the approach marks the iteration from which the value of a variable comes from as a superscript of each variable. After this step, we will have $c^0_{19} = (+\ c^{-1}_{19}\ \text{“}A\text{”})$ and $f^0_{20} = (+\ f^{-1}_{20}\ (+\ c^{-1}_{19}\ \text{“}A\text{”}))$.

The approach continues iterating over the code in R3. c_{19} does not contain any other variables that need to be replaced, but it does contain a self-reference, so the approach adds a τ expression and changes c_{19} in the expression to σ. The result after this step is shown in Equation (4). For f_{20}, the approach replaces c_{19} with its definition shown in Equation (4). Since c_{19} is in the expression of f_{20} with a

-1 superscript, the approach decreases the superscript of all variables in the expression of c_{19} by 1. So, after the replacement, we have:

$$f_{20}^{0} = (+\ \underline{f_{20}^{-1}}\ (+\ (\tau_{c_{19}.R3}^{-1}(+\ \sigma_{c_{19}.R3}^{-2}\ \text{“}A\text{”}))\ \text{“}A\text{”}))$$

Then the approach determines that f_{20} also has a self-reference (underlined), so the approach changes it to a σ, as is shown in Equation (5).

Finally, the approach checks the expression of f_{20} and finds that it contains two τ expressions, the first one is $\tau_{f_{20}.R3}^{0}$, and the second one is $\tau_{c_{19}.R3}^{-1}$. However, these two τ expressions do not represent different loops, they are generated in the same region and represent two related variables in the same loop. So the approach replaces $\tau_{c_{19}.R3}^{-1}$ with $\phi_{c_{19}.R3}^{-1}$. After the replacement, we have the IR of f_{20} as shown in Equation (2). For the other leaf region, R2, its body only defines one variable, e_{33}. The approach represents its IR as shown in Equation (1).

After the approach solves all leaf regions, it solves region R1, which is the parent of R2. R1 is also the body of a loop, it directly defines a variable d_{29} and indirectly defines the variable e_{33} in R2. Similar to R3, in the first step, the approach first replaces all variables with the known representations. Since the approach knows the IR of e_{33} in R2, it directly uses the IR as the expression of e_{33} in R1. Thus, in R1, the approach will have the value of e_{33} the same as in R2, because e_{33} is not changed in R1 and $d_{29} = (+\ d_{29}\ (\tau_{e_{33}.R2}^{0}\ (+\ \sigma_{e_{33}.R2}^{-1}\ \text{“}e\text{”})))$. Then the approach adds superscripts to the internal variable, which is d_{29}, so we have

$$d_{29}^{0} = (+\ d_{29}^{-1}\ (\tau_{e_{33}.R2}^{-1}\ (+\ \sigma_{e_{33}.R2}^{-2}\ \text{“}e\text{”})))$$

Finally, the approach adds a τ expression to d_{29} and we have

$$d_{29} = (\tau_{d_{29}.R1}^{0}\ (+\ \sigma_{d_{29}.R1}^{-1}\ (\tau_{e_{33}.R2}^{-1}\ (+\ \sigma_{e_{33}.R2}^{-2}\ \text{“}e\text{”}))))$$

The expression of d_{29} also contains two τ expressions, but they are from different region and represent different loops.

Finally, the approach analyzes the region R0, which is an MBR. Since it is not a loop body, the approach only replaces the variables with their corresponding expressions. In R0, the approach finds that c_{15}, f_{16}, d_{25}, and e_{26} are also constants and therefore uses their values in the expressions of c_{19}, f_{20}, d_{29}, and e_{44}.

4. EVALUATION

We evaluated our approach by measuring its performance in terms of accuracy and runtime on a mixture of benchmark test cases and real-world Java and Android applications. To provide a baseline for our measurements, we compared our approach against the popular and widely used JSA [9]. Our evaluation addressed the following four research questions:

RQ 1: How accurate is our approach in analyzing strings constructed with various data flows?

RQ 2: How accurate is our approach in handling basic string operations?

RQ 3: How accurate is our approach on real-world applications?

RQ 4: What is the runtime of our approach?

4.1 Implementation

The implementation of our approach, Violist, is written in Java. To extract SSA based representations of an application's code, CGs, and CFGs, we use the Soot [33] analysis framework. We also implemented two string interpreters, described below, that mimic two of the most common approaches to string analysis. For most users of the framework, these two interpreters subsume most string analysis we saw in the literature. Our approach is not limited to these two interpreters though, as the approach can be easily extended to include new interpreters that implement different string operation semantics. In its current state of implementation, Violist can perform string analysis on apps for which either Java or Dalvik bytecode is available. It can be extended to other languages for which it is possible to represent their code in SSA form and generate CGs and CFGs.

Our first interpreter is the String Set Interpreter (SSI). This interpreter is provided a value n that represents the number of times a loop (i.e., a τ) will be unraveled in the IR. Note that this approach is not safe unless n is larger than the maximum loop unraveling or there are no loops in the IR. However, this mimics simple string analyses that do not need loop semantics (e.g., [26]). The output of the SSI interpreter is a set of concrete strings. For example, to interpret the IR for c_{19}, which is shown in Equation (4), with n equal to 3, we would get the set of concrete strings $\{\text{“}AA\text{”}, \text{“}AAA\text{”}, \text{“}AAAA\text{”}\}$.

Our second interpreter is the FSA Interpreter (FSAI). This interpreter approximates the values of strings in the form of FSAs. In general, most of the interpretation semantics for the FSAI are similar to those of the SSI. However, the key difference is that values are not converted to concrete strings, but instead FSAs, and loops are not unraveled. For loops, the FSAI does not unravel τ expressions using an n-bound, instead the FSAI uses the $Widen$ operation proposed by Yu and colleagues [41] to generate a safe model for infinite string values in a loop. In our implementation of FSAI, we leveraged the `Automaton` library used by JSA to model the string outputs. The output of the FSAI is safe due to the approximation process defined by the Yu and colleagues' technique.

4.2 Experiment Protocol

To evaluate the performance of our approach, we measured Violist's accuracy and runtime and compared the results with the publicly available implementation of JSA. For both analyses, we treated all public methods as possible entry points. To measure accuracy, we computed precision and recall of the two analyses against the ground truth. Below, we explain how we identified the ground truth for the different types of applications used in our study and how we compared the ground truth against the models generated by the analyses. All of our experiments were conducted on a Dell XPS-8300 desktop with an Intel i7 processor at 3.4 GHz and 8 GB of memory.

Building Ground Truth: In our evaluation, we used three different types of subjects: the JSA benchmarks, handcrafted test cases, and real-world Android and Java applications. For the first two types, we were able to identify the ground truth by manual analysis of the code, because it was provided as part of the benchmark. However, it was not possible to identify the ground truth for the Android market apps because of their size and the lack of source code.

For these apps, we used a profiling technique to identify a subset of the values that string variables in the apps could assume. To do this, we first randomly selected a set of non-constant string variables in each app that did not include external strings (e.g., user input). We then inserted probes to record the values of these string variables when they were executed. Then we created workloads that traversed all of the discoverable User Interfaces (UIs) and recorded the values of the string variables via the probes. We used these collected values as the ground truth for the string variables. For these apps, the collected values represent a subset of the real ground truth. The implication of this is that there may be less false positives and more false negatives than reported by our results. Thus, by using the profiled subset of the ground truth, we will get a lower bound on the precision and an upper bound on the recall. For both approaches, it is important to note that they provide safe approximations, so we would expect 100% recall. For precision, we do not expect that this calculation of the ground truth would impact one approach more than the other.

Calculation: To calculate precision and recall, we compared the ground truth to the set of strings reported by the two approaches, JSA and Violist. For the JSA and FSAI interpreters, additional steps are needed to compute these values because their string representations may contain cycles, which represent strings of infinite length. For this situation, we followed the Sound and Most Precise (SMP) policy in which we set an upper bound to the length of strings or maximum loop iterations that maintained 100% recall but had the largest precision. For the FSA models generated by JSA and FSAI, we generated strings with length smaller or equal to k, where k is the length of the longest string in the ground truth. For the SSI interpreter, we iterated over the τ expressions for j times, where j is the smallest number of iterations needed by any loop to generate all of the strings in the ground truth.

4.3 RQ 1: Accuracy on Various Types of Data Flow

One of the goals of our approach is to develop a framework that would allow for more accurate modeling and handling of complex data flows, such as inter-procedural manipulation of strings and loops. To evaluate the success of our approach with respect to these goals, we created a test suite that included string value computations based on different types of data flows. In particular, we considered four such types: Branch, Single Loop, Nested Loop, and Inter-procedural. We also defined a fifth type called Circle Loop, which is a single loop containing three variables that depend on each other in a cycle. We combined these five variants to create a mixture of test cases, removing those that did not make sense to combine (i.e., Single Loop, Nested Loop, and Circle Loop). In total, we have eight more types of data flows, which are Branch and Inter-procedural, Single Loop and Branch, Single Loop and Inter-procedural, Nested Loop and Branch, Nested Loop and Inter-procedural, Circle Loop and Branch, Nested and Inter-procedural, and Mix up all. This in total gave us 13 different types of data flow.

Next, we inserted string operations into the different data flows. We broadly categorized all of the string operations into one of three categories and used one function from each as a representative. The first category is Concatenation (Concat), which included all APIs that joined two or more strings. Its representative is StringBuffer.append. The second category is Manipulation (Manip), which include APIs that manipulate a substring of the original string, such as replaceAll, replaceFirst, and substring. Its representative is String.replaceAll. The third category is Converge, which include the APIs that the result will converge after one invocation to the APIs. Its representative is String.trim. For each of the types of data flows, we created one test case for each of the three categories of string operations and one test case that contains all three categories. This gave us 52 total cases.

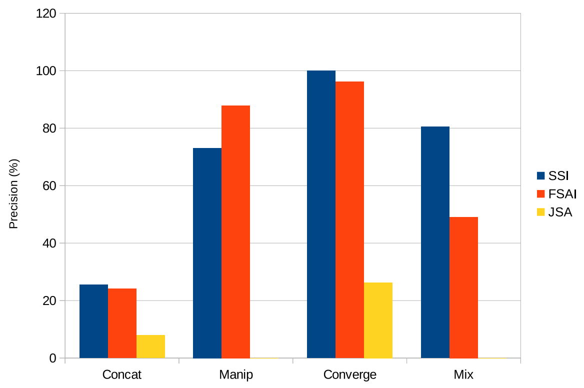


Figure 2: Precision of the three analyses on varied data flows.

Using the process described in Section 4.2, we calculated the precision and recall for each test case for each of the three string analyses. In all cases, the recall for the analyses was 100%, which was expected given our use of the SMP policy. The precision of the three analyses is shown in Figure 2. In this figure, the x-axis shows the names of the four categories of string operations and the individual string analysis. The y-axis shows the average precision for each operation over the different data flows. The average precision of SSI was 70%, FSAI was 64%, and JSA was 7%. From the results, we can see that both of Violist's interpreters achieved precision that was, at least, nine times higher than that of JSA. When we examined the FSAs generated by JSA, this gap was clearly attributable to the increased precision available due to improved loop modeling and context sensitivity. Overall, these results show that our method of handling these types of data flows can result in significant improvements in precision as compared to the popular and widely used JSA.

4.4 RQ 2: Accuracy for Basic String Operations

For the second research question, we looked at the accuracy of Violist and JSA in computing models of the basic string operations without the effect of more complex data flows. In other words, the accuracy of computing string values for a standalone invocation of a string operator. To address this question, we leveraged a set of string analysis benchmarks that were made available as part of the JSA distribution [3]. These test cases cover most of the basic string operations and their usages (e.g., simple branches and loops). To have a meaningful comparison, we removed those test cases that contained operations that were not supported by JSA. We identified these as operations that were implemented by simply modeling them as any string (i.e., ".*"). In total, we had 80 viable test cases. We then ran the three analyses on the test cases and computed their precision and recall.

Both JSA and the two interpreters of Violist achieved 100% recall, which again was expected since they are safe analyses and we used the SMP policy. On average, JSA achieved 76% precision and Violist achieved 71% precision. Both the SSI and FSAI had the same level of precision because the data flow in the test cases was simple enough that the *Widen* operation did not make any difference. Among the 80 test cases, JSA and Violist achieved the same precision in 60 test cases. JSA had a higher precision than Violist in 15 cases while Violist had a higher precision in five cases.

To understand the results better, we inspected the 15 test cases for which JSA had a higher precision in more detail. It turns out that besides string analysis, JSA also performs basic string constraint solving for string constants. For example, if there is a branch statement that contains a constraint on a string, JSA can augment its results with this information. Our approach does not refine the results in this way, so it had lower precision for these test cases. We also inspected the test cases for which Violist had a higher precision. We found that Violist had a higher precision for these because of its ability to more precisely model inter-procedural analysis.

Overall, we believe that the results of this research question are neutral. For most of the implemented string operations, our approach and JSA were equivalent in terms of precision. For those in which JSA was better, we found that the reason for this was that JSA also performed basic analysis of string constraints in branches. Although our analysis could not handle these, the architecture of our approach would allow a user to incorporate these semantics into an interpreter. For example, by attaching string constraints to each region of the Region Based Techniques (RBT) and then using a string constraint solver to account for their semantics. Additionally, these results discount a threat to validity of RQ1; namely that the observed increase in precision was due to differences in the implementation of the basic string operations used in the various data flows.

4.5 RQ 3: Accuracy on Realistic Apps

In the third research question, we looked at the performance of the string analyses on realistic string variables and programs. We selected five realistic apps from different sources. The name, description, and size of the apps are listed in Table 1. The unit of size is the number of bytecode instructions. Among our apps, Reflection is one of JSA's benchmark apps [2]; BookStore is from our previous testbed [1]; and apps a2z, add.me.fast, and pipefilter are three mobile apps randomly chosen from the Android Play Market. We only used one of the JSA benchmarks because we were not able to locate older versions of libraries required for their compilation. In total, over these five apps, we recorded 133 different string variables.

We ran all three analyses on the subject apps and computed the precision and recall as described in Section 4.2. As with the previous experiment, the recall was 100% for all analyses for all apps. For the three mobile apps, JSA did not terminate before running out of memory, so we were unable to compute its accuracy. We investigated this issue further and believe the reason is due to the fact that JSA uses a variable-pair-based method to do the global inter-procedural aliasing analysis. This method has an $O(n^2)$ memory complexity, where n is the number of variables across the whole application. In our experiments, Android applications generally have more than 100,000 variables. Thus, we hypothesize that the failure is due to the increased expense of the global alias analysis when running over an app that makes extensive use of a large framework (i.e., the Android API).

Table 1: Description of subject apps

App	Description	Size
Reflection	Java reflection benchmark in JSA	7,156
BookStore	A JSP web app	24,305
a2z	An Android conference app	74,700
add.me.fast	A social network app	39,330
pipefilter	An information checking app	16,626

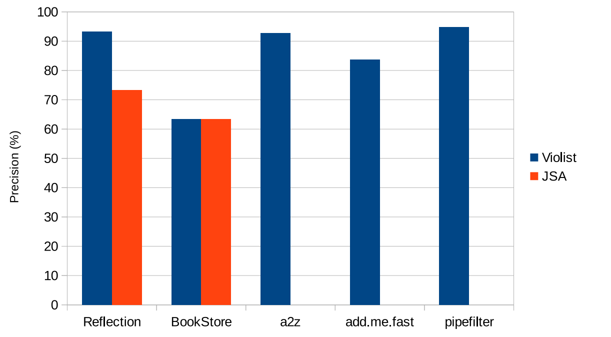


Figure 3: The precision of market apps

Figure 3 shows the precision measured in the experiment. Both SSI and FSAI had the same precision results, so they are shown together. For the two apps measured, Violist was either equal to or higher in terms of precision. Across all apps, Violist had an average precision of 86% while JSA had an average precision of 68% on its two working apps.

Overall, these are positive results for Violist. It was able to achieve, on average, a higher level of precision than JSA, thus showing that its results from RQ1 and RQ2 carry over to real world applications. Furthermore, the results highlight the general applicability of Violist, as it was able to accurately analyze apps written for Android as well.

4.6 RQ 4: Analysis Runtime

To answer this question, we compared the execution time of Violist's two interpreters against JSA. For each of the experiments we reported in RQ 1–3, we repeated the experiment ten times and recorded the execution time for each. The results of these timing measurements are shown, in milliseconds, in Table 2. Due to the large number of test cases in RQ 1 and RQ 2, we only reported the average time cost for each of their test cases. In our results, we subtracted out the time consumed by the Soot APIs (for generating call graphs and control flow graphs) from the timing measurements, since this was strictly time consumed by Soot and was the same for both.

The results show that for all categories of tests, Violist was significantly faster. The FSAI interpreter averaged a 75% decrease in runtime versus JSA and the SSI interpreter averaged a 94% decrease versus JSA. As with the previous RQ, we hypothesize that the cost of performing the global alias analysis is responsible for the high runtime. Within

Table 2: Time cost of Violist vs. JSA

Case Name	SSI (ms)	FI (ms)	JSA (ms)
RQ 1 test cases	13	22	119
RQ 2 test cases	35	306	401
Reflection	101	147	54,762
BookStore	56,607	202,257	7,371,030
a2z	455,772	457,248	N/A
add.me.fast	34	174	N/A
pipefilter	16,913	17,083	N/A

the results for Violist, the SSI interpreter was clearly faster. The reason for this is that the *Widen* operation for FSAs is a relatively expensive operation. Overall, these are positive results for our approach and show that our approach can significantly outperform JSA in terms of analysis runtime.

5. RELATED WORK

The most similar work to ours is Nguyen and colleagues' D-model [29], which defines a tree-based representation for symbolic, string-based values originating from symbolic execution of PHP expressions. Our approach is different from this model in the following aspects. First, their approach addresses the problem of estimating multiple outputs of client-side pages while our approach targets a more general problem, which is estimating the possible values of a string variable at a given program point. Second, their model only handles string concatenation, while ours supports all of the string operations in the Java API. Third, their model only unravels loops once, while our approach can unravel loops either a set n number of times or based on additional code analyses.

Another closely related work is JSA [9], which uses flow graphs to produce FSAs that represented the possible string values of Java string variables. Compared with the flow graphs of JSA, the IR of Violist has two main advantages. The first advantage is that it allows users to specify the iteration bound for loops while JSA assumes the bound is infinite. The second advantage is that it uses a summary based technique so that it can provide context sensitive inter-procedural analysis and avoid expensive global aliasing analysis.

Other approaches also exploit symbolic execution to perform string analysis for Java [32] and Javascript [31]. Since these techniques use symbolic execution, they may not be readily scalable to complex programs (e.g., those containing thousands of branches). Other research related to symbolic execution based string analysis uses a constraint solver [10, 34, 35, 42, 13, 7] or a decision procedure [23, 25] to decide whether the string constraints are satisfiable or not. These approaches are generally concerned with string constraint solving as part of a symbolic execution based approach to string analysis, as opposed to our more general focus of a static string analysis framework.

The automaton-based method is also widely used to perform string analysis. Minamide [28] used transducers (i.e., multi-track automata) to do PHP string output analysis. Other techniques [41, 8, 5] introduced the widening operator for loops to guarantee the convergence of symbolic reachability analysis. A drawback of these automaton-based approaches is that they overestimate the iteration bound for loops; generally assuming it is infinite. Our approach can do this as well, but also offers a flexible way to incorporate user provided information (e.g., via additional program analyses) to more accurately model loop bounds, which can help increase the precision of the estimated string values.

Based on string analysis, researchers have made great progress in many verification areas. For example, locating buffer overflow errors due to string manipulations [36, 11, 12], detecting SQL query type errors [18, 17, 37], SQL injection attacks [38, 39, 19, 14], cross-site scripting vulnerabilities [24, 40], HTML errors [29, 21], identifying incorrect or incomplete sanitization [6], characterizing database interactions [27], improving test case generation [20], performing permission analysis [16], and understanding runtime behavior of DEB systems [15]. All of these techniques exploit existing string analysis directly or indirectly to attain different research goals. Our work proposes a flexible string analysis framework, which could be used by these approaches to improve their results.

6. THREATS TO VALIDITY

To ensure that we can have a fair comparison to JSA, we selected our benchmarks from three different sources. These included apps from JSA's benchmarks or that had been analyzed by JSA in prior work [19] as well as test cases we had created. Although test cases created by our group could introduce bias, there were no benchmarks available that included the more complex data flows we wanted to test.

A general threat to the validity of RQ3 is our mechanism for establishing ground truth. If we were incomplete, we would have an upper bound on recall and a lower bound on precision. As we expect both analyses to have 100% recall, this primarily would affect precision. A change in precision would affect one of our conclusions from RQ3, but would not change our larger finding that Violist could more easily scale up for larger applications than JSA.

7. CONCLUSION

String analysis is a fundamental technique for many verification techniques, such as HTML output analysis and SQL injection detection. In this paper, we propose a general framework, Violist, for string analysis that allows for modeling string variables with high accuracy. In particular, Violist allows for more precise modeling of loops and inter-procedural string manipulations; while providing users with an easy way to use custom string semantics and scale up for large applications. In our evaluation, Violist achieved high precision and recall in complex market apps. Compared to JSA, the state-of-the-art alternative solution, Violist achieved over nine times higher precision on complex data flows with the same recall and a runtime speed several orders of magnitude faster. These results show that our approach can potentially improve the performance of other verification techniques that are built based on string analysis techniques.

8. ACKNOWLEDGMENTS

This work was supported by NSF grant CCF-1321141.

9. REFERENCES

[1] http://www-bcf.usc.edu/∼halfond/testbed.html.
[2] http://www.brics.dk/JSA/dist/jsa-benchmarks.tar.gz.

[3] http://www.brics.dk/JSA/dist/string-test.tar.gz.

[4] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.

[5] M. Alkhalaf, T. Bultan, and J. L. Gallegos. Verifying Client-side Input Validation Functions Using String Analysis. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 947–957, Piscataway, NJ, USA, 2012. IEEE Press.

[6] D. Balzarotti, M. Cova, V. Felmetsger, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. Saner: Composing Static and Dynamic Analysis to Validate Sanitization in Web Applications. In *Security and Privacy, 2008. SP 2008. IEEE Symposium on*, pages 387–401, May 2008.

[7] N. Bjørner, N. Tillmann, and A. Voronkov. Path Feasibility Analysis for String-Manipulating Programs. In *Proceedings of the 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems: Held As Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009,*, TACAS '09, pages 307–321. Springer-Verlag, Berlin, Heidelberg, 2009.

[8] T.-H. Choi, O. Lee, H. Kim, and K.-G. Doh. A Practical String Analyzer by the Widening Approach. In N. Kobayashi, editor, *Programming Languages and Systems*, volume 4279 of *Lecture Notes in Computer Science*, pages 374–388. Springer Berlin Heidelberg, 2006.

[9] A. S. Christensen, A. Møller, and M. I. Schwartzbach. Precise Analysis of String Expressions. In *Proceedings of the 10th International Conference on Static Analysis*, SAS'03, pages 1–18, Berlin, Heidelberg, 2003. Springer-Verlag.

[10] L. De Moura and N. Bjørner. Z3: An Efficient SMT Solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[11] N. Dor, M. Rodeh, and M. Sagiv. Cleanness Checking of String Manipulations in C Programs via Integer Analysis. In P. Cousot, editor, *Static Analysis*, volume 2126 of *Lecture Notes in Computer Science*, pages 194–212. Springer Berlin Heidelberg, 2001.

[12] N. Dor, M. Rodeh, and M. Sagiv. CSSV: Towards a Realistic Tool for Statically Detecting All Buffer Overflows in C. *SIGPLAN Not.*, 38(5):155–167, May 2003.

[13] X. Fu and C.-C. Li. A String Constraint Solver for Detecting Web Application Vulnerability. In *SEKE*, pages 535–542, 2010.

[14] X. Fu, X. Lu, B. Peltsverger, S. Chen, K. Qian, and L. Tao. A Static Analysis Framework For Detecting SQL Injection Vulnerabilities. In *Computer Software and Applications Conference, 2007. COMPSAC 2007. 31st Annual International*, volume 1, pages 87–96, July 2007.

[15] J. Garcia, D. Popescu, G. Safi, W. G. Halfond, and N. Medvidovic. Identifying message flow in distributed event-based systems. In *Proceedings of the Symposium on the Foundations of Software Engineering (FSE)*, August 2013.

[16] E. Geay, M. Pistoia, T. Tateishi, B. Ryder, and J. Dolby. Modular string-sensitive permission analysis with demand-driven precision. In *Software Engineering, 2009. ICSE 2009. IEEE 31st International Conference on*, pages 177–187, May 2009.

[17] C. Gould, Z. Su, and P. Devanbu. JDBC checker: a static analysis tool for SQL/JDBC applications. In *Software Engineering, 2004. ICSE 2004. Proceedings. 26th International Conference on*, pages 697–698, May 2004.

[18] C. Gould, Z. Su, and P. Devanbu. Static checking of dynamically generated queries in database applications. In *Software Engineering, 2004. ICSE 2004. Proceedings. 26th International Conference on*, pages 645–654, May 2004.

[19] W. G. Halfond and A. Orso. AMNESIA: Analysis and Monitoring for NEutralizing SQL-Injection Attacks. In *Proceedings of the International Conference on Automated Software Engineering*, pages 174–183, Long Beach, California, USA, November 2005.

[20] W. G. Halfond and A. Orso. Improving Test Case Generation for Web Applications Using Automated Interface Discovery. In *Proceedings of the Symposium on the Foundations of Software Engineering*, Dubrovnik, Croatia, September 2007.

[21] W. G. Halfond and A. Orso. Automated Identification of Parameter Mismatches in Web Applications. In *Proceedings of the Symposium on the Foundations of Software Engineering*, November 2008.

[22] W. G. J. Halfond, S. R. Choudhary, and A. Orso. Penetration Testing with Improved Input Vector Identification. In *Proceedings of the International Conference on Software Testing, Verification, and Validation*, pages 346–355, Denver, Colorado, USA, Apr. 2009.

[23] P. Hooimeijer and W. Weimer. A Decision Procedure for Subset Constraints over Regular Languages. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 188–198, New York, NY, USA, 2009. ACM.

[24] N. Jovanovic, C. Kruegel, and E. Kirda. Pixy: a static analysis tool for detecting Web application vulnerabilities. In *Security and Privacy, 2006 IEEE Symposium on*, pages 6 pp.–263, May 2006.

[25] A. Kiezun, V. Ganesh, P. J. Guo, P. Hooimeijer, and M. D. Ernst. HAMPI: A Solver for String Constraints. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis*, ISSTA '09, pages 105–116, New York, NY, USA, 2009. ACM.

[26] D. Li, A. H. Tran, and W. G. J. Halfond. Making Web Applications More Energy Efficient for OLED Smartphones. In *Proceedings of the International Conference on Software Engineering (ICSE)*, June 2014.

[27] E. Martin and T. Xie. Understanding Software Application Interfaces via String Analysis. In *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, pages 901–904, New

York, NY, USA, 2006. ACM.

[28] Y. Minamide. Static Approximation of Dynamically Generated Web Pages. In *Proceedings of the 14th International Conference on World Wide Web*, WWW '05, pages 432–441, New York, NY, USA, 2005. ACM.

[29] H. V. Nguyen, H. A. Nguyen, T. T. Nguyen, and T. N. Nguyen. Auto-locating and Fix-propagating for HTML Validation Errors to PHP Server-side Code. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, ASE '11, pages 13–22, Washington, DC, USA, 2011. IEEE Computer Society.

[30] G. Safi, A. Shahbazian, W. G. Halfond, and N. Medvidovic. Detecting event anomalies in event-based systems. In *Proceedings of the 10th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*, September 2015. To Appear.

[31] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song. A Symbolic Execution Framework for JavaScript. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP '10, pages 513–528, Washington, DC, USA, 2010. IEEE Computer Society.

[32] D. Shannon, S. Hajra, A. Lee, D. Zhan, and S. Khurshid. Abstracting Symbolic Execution with String Analysis. In *Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION, 2007. TAICPART-MUTATION 2007*, pages 13–22, Sept 2007.

[33] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot - a java bytecode optimization framework. In *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research*, CASCON '99, pages 13–. IBM Press, 1999.

[34] M. Veanes, N. BjÃÿrner, and L. de Moura. Symbolic Automata Constraint Solving. In C. FermÃijller and A. Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, volume 6397 of *Lecture Notes in Computer Science*, pages 640–654. Springer Berlin Heidelberg, 2010.

[35] M. Veanes, P. d. Halleux, and N. Tillmann. Rex: Symbolic Regular Expression Explorer. In *Proceedings of the 2010 Third International Conference on Software Testing, Verification and Validation*, ICST '10, pages 498–507, Washington, DC, USA, 2010. IEEE Computer Society.

[36] D. Wagner, J. S. Foster, E. A. Brewer, and A. Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *In Network and Distributed System Security Symposium*, pages 3–17, 2000.

[37] G. Wassermann, C. Gould, Z. Su, and P. Devanbu. Static Checking of Dynamically Generated Queries in Database Applications. *ACM Trans. Softw. Eng. Methodol.*, 16(4), Sept. 2007.

[38] G. Wassermann and Z. Su. An analysis framework for security in Web applications. In *In Proceedings of the FSE Workshop on Specification and Verification of Component-Based Systems (SAVCBS 2004*, pages 70–78, 2004.

[39] G. Wassermann and Z. Su. Sound and Precise Analysis of Web Applications for Injection Vulnerabilities. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 32–41, New York, NY, USA, 2007. ACM.

[40] G. Wassermann and Z. Su. Static Detection of Cross-site Scripting Vulnerabilities. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, pages 171–180, New York, NY, USA, 2008. ACM.

[41] F. Yu, T. Bultan, M. Cova, and O. Ibarra. Symbolic String Verification: An Automata-Based Approach. In K. Havelund, R. Majumdar, and J. Palsberg, editors, *Model Checking Software*, volume 5156 of *Lecture Notes in Computer Science*, pages 306–324. Springer Berlin Heidelberg, 2008.

[42] Y. Zheng, X. Zhang, and V. 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*, ESEC/FSE 2013, pages 114–124, New York, NY, USA, 2013. ACM.