World Academy of Science, Engineering and Technology
International Journal of Information and Communication Engineering
Vol:14, No:12, 2020

Automata-Based String Analysis for Detecting Malware in Android Programs

Assad Maalouf, Lunjin Lu, James Lynott

Abstract—We design and implement a precise model of string operations using finite state machine transformers and state transformers to approximate the values string variables can take throughout the execution of the program. We use our model to analyze Android program string variables. Our experimental results show that our string analysis is very efficient at detecting the contextual effect of string operations on the string variables. Our model proved to be very useful when it came to verifying statements about the string variables of the program.

Keywords—Abstract interpretation, android, static analysis, string analysis.

I. Introduction

STRING analysis plays an important role in detecting vulnerabilities of Android applications. In Android, a bridge between JavaScript and Java code allows JavaScript code to invoke privileged system operations. An attacker can exploit this vulnerability by injecting malicious code into a string passed to a web browser component hosted by the mobile app. An analyzer can detect those attacks by checking if the final string can contain an attack pattern. JavaScript command injection attacks can be detected by the presence of special characters in the untrusted input, the characters that define a script tag, the less than character, the greater than character, and the keyword `script`. Code injection attacks are the most popular forms of attacks in distributed applications [1]. The confidentiality and integrity of an application can be compromised by attacks caused by strings originating from untrusted sources injected with malicious scripts. The malicious script reaches a sensitive operation and executes with the same permissions of the application, which can compromise the system. String analysis also serves as a building block of more complex analyses. In taint analysis, a precise string analysis provides useful information for detecting the contextual effect of a string operation on taint propagation. For instance, confidential information can be represented with a pattern or a regular expression and the analyzer can detect if the final expression still contains the tainted pattern, or if it was sanitized by the program.

The previous state-of-the-art string analyzers of Java programs such as JSA [2] are not precise enough as they do not abstract all concrete string operations precisely. For instance, JSA's abstract string comparison operations such as `contains`, `equals`, `startsWith`, `endsWith` and `isEmpty` are precise only when their abstract operands are constants. Furthermore, its abstract `insert`, `substring` and `replace` operations ignore numerical index operands. In this paper, a precise string analysis is presented. Inspired by [3], we use the domain of finite state automata as the abstract string domain and finite state automaton transformers as abstract string operations. The main novelty of this work is twofold: (a) designing abstract operations that approximate Android string operations including those that are not addressed in [3], and (b) applying numerical analysis to improve approximations of string operations with numerical operands, and (c) precise analysis of conditional expressions based on (a) and (b). The result is a more precise analyzer for Android applications. Our experimental results show that our string analysis is very efficient at detecting contextual effect of string operations on the string variables.

The rest of the paper is organized as follows. In Section II, we present a motivation example. In Section III, we discuss the related work. In Sections IV, V and VI we present our string analysis. In Section VII, we present the experimental results and we conclude in SSection VIII.

II. Motivating Example

To showcase the benefits of a precise string analysis on taint propagation, consider the following program code:

```
1 String s = TelephonyManager.getDeviceId(); //source of sensitiv
2 String t = "neutral";
3 while (Math.random() ≤ 0.5) t = t.concat(s);
4 if (!t.contains(s))(new ConnectionManager()).publish(t); //sink
```

Fig. 1 Code Snippet

The broadcast instruction at line 4 is constrained with the condition that the sensitive piece of information that was read at line 1 does not occur in the variable being published. State of the art static taint analyzers such as FlowDroid [4], treat it as a leak because their approximations to string comparison operations such as `contains` are too coarse. Our string analysis precisely models string comparison operations as abstract store transformers and allows us to conclude that the variable being published does not contain any sensitive information.

III. Related Work

Violist [5] is a framework for string analysis of Java and Android applications. Violist gives the user the ability to define custom semantics for the string operations to closely model their effect in the problem domain being analyzed. Violist

Assad Maalouf is with the Oakland University, United States (e-mail: amaalouf@oakland.edu).

relies on symbolic execution; it separates the representation from the interpretation of string operations. Violist models relationships between string variables and string operations using an intermediate representation and then interprets these operations with the provided custom semantics to compute the effect of the string system. Violist provides the user with complex constructs for modeling loops and nested loops as well as for tracking dependencies between the variables affected by those loops. The custom semantics can then approximate the behavior of those loops using finite loop unraveling or using widening to approximate values of infinite strings.

Epicc [6] analyses the flow of information between different components of Android applications. Epicc reduces the problem of inter-component communication (ICC) of Android applications to an instance of an IDE [7] problem for finding ICC vulnerabilities with high accuracy. Epicc uses string analysis to detect communication from one component to another and approximates the parameters of this communication to determine the possible receivers of the intent as well as the data being exchanged. In Epicc, distributive environment transformers model the effects of system method calls on the intents and the data they carry. The problem is solved using Heroes [7] and Soot [8].

JSA [2] is a String analyzer for Java programs. JSA uses a regular language to compute a superset of the values a string variable can take. JSA is used for verifying properties of strings that are dynamically computed, such as dynamically generated XML strings and dynamically generated SQL queries. JSA transforms the string system into a context-free grammar (CFG), with a production for each string expression. Modeling string operations using finite state machine transformers, the CFG is widened into a regular language, approximating the state space of the program. JSA only considers strings and string manipulating operations that occur in the program, abstracting everything else away and intelligently extracting a regular language that over approximates the language of the original context-free grammar. JSA models the operations on strings using finite state machine transformers. Operations such as `replace`, and `substring` are defined on automatons to produce a machine whose language over approximates the set of strings that occur as a result of this concrete operation. The modeling of the abstract operations in JSA is not precise enough and quickly falls into broader over approximations. JSA ignores the operands of most of the string operations and quickly loses its precision. For example, JSA does not account the for numerical operands of abstract operations such as `substring` and `insert`. The model JSA uses for `string.replace` cannot handle the case where the string operands are finite state machines.

FlowDroid [4] [9] is a context sensitive taint flow analyzer of Android components, it is able to detect flow from a private piece of information into a public sink. In FlowDroid taint analysis becomes an IDE [7] problem and is resolved using Heroes [7] and Soot [8] frameworks for static analysis.

While FlowDroid [4] is value insensitive and relies completely on Soot modules to optimize the call graph generation and to eliminate unreachable code. Epicc [6] benefits from a simple constant propagation analysis which allows the tool to evaluate simple string expressions that evaluate to a literal. The analysis approximates the value of the string expression to be any string when the result no longer evaluates to a constant string literal or when the expression is too complex to be evaluated by the analysis. The results of string analysis are used to resolve intents and to approximate receivers of inter component communication, among other usages. The string analysis Epicc performs is not precise enough and results in broader approximations and lots of false positives where the tool assumes a communication can occur, whereas in practice it does not. Yu et al. [3] does not model string operations such as `substring` and `insert` and their modeling of string conditional constructs is very limited. They only handle conditional constructs such as equality of strings and `isEmpty`. Our modeling of the conditional constructs on strings is more elaborate and more precise. We handle conditional string expressions such as `contains`, `startsWith`, `endWith` in addition to `equals` and `isEmpty`. We also handle the negation, the conjunction and the disjunction of string conditional expressions.

There has been much work on string analysis of programs in Java / Android and other programming languages [2], [5], [6], [10]- [16]. The abstract domains proposed for stings analysis range from constant propagation [10], to multitrack finite state machines [16], providing different trade-offs between precision and cost of analysis.

Yu et al. [16] define the lattice of the multitrack deterministic finite state machine. A multitrack deterministic finite state machine transits on multiple string variables rather than on just one string variable, like a single track deterministic finite state machine does. Multi-track automatons are used to improve the precision of string analysis algorithms. Multi-track automatons provide a relational form of tracking, which allows the expression of a more elaborate form of verification that prevents many of the errors that cannot be expressed with single track automatons.

Amadini et al. [10] use abstract interpretation to combine simple abstract domains into a hybrid domain resulting in an original and precise analysis of string values. The abstract domain is defined as the product of multiple simple well-known existing abstract domains, such as the String Set domain [10], the Constant String domain [10], the Character Inclusion domain [11], the Prefix, Suffix domain [11] and the String Hash domain [12] as well as JavaScript specific domains from [13]- [15].

The String Set Domain SS_k can represent at most k strings precisely before losing its precision. Its abstract domain is the subsets of strings whose cardinality is at most k augmented with a top element. The abstraction function maps a subset of strings into itself when its cardinality is less than or equal to k or into the top of the lattice otherwise. The concretization function maps a subset of size at most k into itself and the top of the lattice into the set of all strings. The Constant string domain or CS is the special case of string subset domain where the number of strings being tracked is exactly one.

In the String Set domain, any string operation that results in a set that exceeds the limit k, the maximal number of strings being tracked, results in a loss of information. In the CS domain, a change in the value of a variable being tacked causes the analysis to lose its information. Consider the example of a set of concrete string values `SS={"relational", "rational"}`

The abstraction of the set `SS` in SS_2 is the set itself and the abstraction of the set `SS` in CS is the top of the lattice.

In the Character Inclusion domain, CI, an element of the abstract domain is a pair $< l, u >$ where l and u are subsets of characters. The abstraction of a set S of words from Σ^* is the pair $< l, u >$, where l is the set of characters that must occur in any w from S, and u is the set of characters that may occur. The concretization of a pair $< l, u >$ is the set of words w that contain all the characters of l and some of the character of u.

The abstraction of the set `SS` in CI is the set `{{"r","a","t","i","o","n","l"},{"r","a", "t","i","o","n","l","e"}}`.

Interval domains ignore the structure of the concrete string being approximated. A word is represented with the characters that occur in it without regard to the order in which they occur. The result is a computationally cheap model that can produce useful information.

In the Prefix Suffix domain, an abstract value is a pair of strings $< p, s >$ that represents all the strings w that start with p and end with s. The abstraction of a set of words S is the pair of words $< p, s >$ where p is the longest common prefix of all the words of S and s is the longest common suffix of all the words of S. The abstraction of the set `SS` in PS is the set `{"r","ational"}`.

The String Hash domain, SH, uses a hash function to map a word or a subset of words into an integer, integer range, or a bucket. In the String Hash domain, a word maps to the sum of the integer ASCII codes of the characters that occur in it. The SH domain is useful for easily inferring inequality of strings. An empty intersection of the hash buckets of two sets of words allows us to conclude that the two sets contain different words.

IV. The Regular Abstract Domain for String Analysis

We use the domain of finite state automata to approximate the sets of strings that variables might take during program execution, because it captures string properties sufficiently precisely and it admits sufficiently efficient implementation.

In the Regular Abstraction a Galois Connection is established between the set of concrete strings and the set of regular languages. The concrete domain $< \wp(\Sigma^*), \subseteq >$ is the power set of strings ordered by subset inclusion. A language is regular *iff* it can be accepted by some finite state machine M. The set of all regular languages over a given alphabet Σ can be represented by the set of all deterministic finite state machines DFA over Σ. Let $M_1, M_2 \in DFA$. Define $M_1 \leq M_2 \iff L(M_1) \subseteq L(M_2)$ where $L(M)$ is the regular language accepted by M . The relation $\leq$ is a preorder and the relation $\equiv$ where $M_1 \equiv M_2 \iff M_1 \leq M_2 \wedge M_2 \leq M_1$ is an equivalence relation. Denote by $[M]_\equiv$ the equivalence class of $\equiv$ containing M and define $[M_1]_\equiv \sqsubseteq [M_2]_\equiv \iff L(M_1) \subseteq L(M_2)$. $< DFA_{/\equiv}, \sqsubseteq >$ is a complete lattice where $DFA_{/\equiv}$ is the quotient of DFA with respect to $\equiv$. In the sequel, we shall use arbitrary member of an equivalence class of $\equiv$ to represent that class since all the abstract operations in our string analysis are homomorphisms with respect to $\equiv$. Let $\mathbf{0}$, $\mathbf{1}$ and $\mathbf{\Lambda}$ be constant DFAs such that $L(\mathbf{0}) = \emptyset$, $L(\mathbf{1}) = \Sigma^*$ and $L(\mathbf{\Lambda}) = \{\epsilon\}$ where ϵ is the empty string. The meet of two DFAs denoted by $\sqcap$, is their intersection and the join is their union.

The abstraction function $\alpha : \wp(\Sigma^*) \longrightarrow DFA_{/\equiv}$ is defined as follows: $\alpha(S) = \sqcap_{\{M | S \subseteq L(M)\}}$. $\alpha(S)$ is the smallest of all machines that accept S. The concretization function $\gamma : DFA_{/\equiv} \longrightarrow \wp(\Sigma^*)$ is defined as follows: $\gamma(M) = L(M)$. $\gamma(M)$ is the set of all words accepted by the machine M. The following equivalence can be easily proved and $< \alpha, \gamma >$ is a Galois connection.

$S \subseteq \gamma(M) \iff \alpha(S) \sqsubseteq M$ where $S \in \wp(\Sigma^*)$ and $M \in DFA_{/\equiv}$

V. Branch Sensitivity

Concrete operations on strings fall into two categories: those that return string values and those that return boolean values. In this section present the abstract operations that approximate those concrete operations that return boolean values. They are abstract store transformers. In the next section we present the abstract operations that approximate those concrete operations that return string values.

We define and implement rules to restrict the incoming abstract store to a given condition and apply constraints on the variables in the condition accordingly. Let Var be a denumerable set of string variables. The set of the abstract stores is $\Pi = Var \longrightarrow DFA$. A string expression $e \in SExp$ is either a string literal $\ell \in \Sigma^*$, or a string variable $v \in Var$, or $e_1.concat(e_2)$, or $e_1.substring(i, j)$ or $e_1.replace(e_2, e_3)$, or $e_1.insert(e_2, i)$ where $e_1, e_2, e_3 \in SExp$ and i, j are integer constants. A conditional expression $c \in BExp$ on string variables is either a primitive conditional expression, or a negation of a conditional expression, or a conjunction of two conditional expressions $c_1 \&\& c_2$ or a disjunction of two conditional expressions $c_1 || c_2$ with $c_1, c_2 \in BExp$. A primitive conditional expression is either $e_1.equals(e_2)$ or $e_1.contains(e_2)$ or $e_1.startsWith(e_2)$ or $e_1.endsWith(2_2)$ or $e1.isEmpty()$ where $e_1, e_2 \in SExp$.

The semantic functions for branch sensitivity make use of the following helper functions: substrings, prefixes, suffixes and complement: $DFA \longrightarrow DFA$. These helper functions take an automaton as input and produce the automaton that accepts the substrings, prefixes, suffixes and complement of the original automaton, respectively. Substrings takes an automaton as input and constructs a machine that accepts any word that can occur as a substring of a word accepted by the original machine. A new initial and a new final state are added to the original machine. An epsilon transition from the new initial state is added to every other state. An epsilon transition is added from every state in the original machine to the new final state. The machine is then determinized. Prefixes takes an automaton as input and constructs a machine that accepts any word that can occur as a prefix of a word accepted by the original machine. A new final state is added to the original machine and an epsilon transition is added from every state to the new final state. The machine is then determinized. Suffixes takes an automaton as input and constructs a machine that accepts any word that can occur as a suffix of a word accepted by the original machine. A new initial state is added to the original machine and an epsilon transition is added from the new initial state to every other state in the original machine. The machine is then determinized. Concat $DFA \longrightarrow (DFA \longrightarrow DFA)$, is a binary operator that takes two automata and returns their concatenation. By $concat(M_1, M_2, M_3)$, we mean $concat(M_1, concat(M_2, M_3))$.

Branch sensitivity is realized through three semantics functions. Semantic function $\mathscr{E} : SExp \longrightarrow (\Pi \longrightarrow DFA)$ evaluates a string expression against an abstract store and returns an abstract string. Semantic function $\mathscr{C} : BExp \longrightarrow (\Pi \longrightarrow \Pi)$ restricts the variables in a condition and returns an abstract store. Semantic function $\mathscr{A} : SExp \longrightarrow DFA \longrightarrow (\Pi \longrightarrow \Pi)$ restricts the variables in an expression so that its value is described by a given abstract string. It returns an abstract store. These semantic functions are defined by 3-29.

Some explanations are in order. Take the example of (9) where the conditional expression $e_1.equals(e_2)$ is to be restricted against $\pi \in \Pi$. The expressions e_1 and $e_2 \in SExp$ are two string expressions and π is the incoming abstract store. We start by evaluating the expressions e_1 and e_2 separately in the incoming abstract store π, Let $a_1 = \mathscr{E}[[e_1]]\pi$ and $a_2 = \mathscr{E}[[e_2]]\pi$. We then restrict the variables in e_1 such that e_1 is a string accepted by a_2 and we restrict the variables in e_2 such that e_2 is a string accepted by a_1. Let $\pi_1 = \mathscr{A}[[e_1]]a_2\,\pi$ and $\pi_2 = \mathscr{A}[[e_2]]a_1\,\pi$. Then we take the meet of the resulting stores, the results that satisfy both restrictions simultaneously. We have $\mathscr{C}[[e_1.equals\,(e_2)]]\pi = \pi' = \pi_1 \sqcap \pi_2$.

Take the example of (10) where the conditional expression $e_1.contains(e_2)$ is to be restricted against π. We start by evaluating the expressions e_1 and e_2 separately in the incoming abstract store π, Let $a_1 = \mathscr{E}[[e_1]]\pi$ and $a_2 = \mathscr{E}[[e_2]]\pi$. We then construct the machine $a'_2 = concat\,(\mathbf{1}, a_2, \mathbf{1})$. a'_2 accepts the words where words from $L(a_2)$ can occur as substrings. We also construct the machine $a'_1 = substrings\,(a_1)$. a'_1 accepts words that occur as substrings of words from $L(a_1)$. We then restrict the variables in e_1 such that e_1 is a string accepted by a'_2 and we restrict the variables in e_2 such that e_2 is a string accepted by a'_1. Let $\pi_1 = \mathscr{A}[[e_1]]a'_2\,\pi$ and $\pi_2 = \mathscr{A}[[e_2]]a'_1\,\pi$. Then we take the join of the resulting stores, the results that satisfy both restrictions simultaneously. We have $\mathscr{C}[[e_1.contains\,(e_2)]]\pi = \pi' = \pi_1 \sqcap \pi_2$

In the case of (11) the conditional expression $e_1.startsWith(e_2)$ is to be restricted against π. Let a_1 and a_2 be the finite state machines resulting from the evaluation of e_1 and e_2 respectively, in π. We construct the automaton $a'_2 = concat\,(a_2, \mathbf{1})$. a'_2 is a machine that accepts the words that start with words from $L(a_2)$. We also construct the automaton $a'_1 = prefixes\,(a_1)$. a'_1 is a machine that accepts words that occur as prefixes of words from $L(a_1)$. We then restrict the variables in e_1 so that its value is accepted by a'_2 and we restrict the variables in e_2 so that its value is accepted by a'_1. We then take the results that satisfy both restrictions simultaneously.

In the case of (12), the conditional expression $e_1.endsWith(e_2)$ is to be restricted against π. Let a_1 and a_2 be the finite state machines resulting from the evaluation of e_1 and e_2 respectively, in π. We construct the automaton $a'_2 = concat\,(\mathbf{1}, a_2)$. a'_2 is a machine that accepts the words that end with words from $L(a_2)$. We also construct the automaton $a'_1 = suffixes\,(a_1)$. a'_1 is a machine that accepts words that occur as suffixes of words from $L(a_1)$. Next, we restrict the variables in e_1 so that its value is accepted by a'_2 and we restrict the variables in e_2 so that its value is accepted by a'_1. We then take the results that satisfies both restrictions simultaneously.

$$\begin{aligned} SExp ::= s|v|e_1.concat\,(e_2)\,|e_1.substring\,(i,j) \\ |replace\,(e_1,e_2,e_3)\,|e_1.insert\,(e_2,i) \end{aligned} \tag{1}$$

where s is a string literal, $v \in Var$ is a string variable, $e_i \in SExp$ and i ,j are integer constants

$$\begin{aligned} BExp ::= e_1.equals\,(e_2)\,|e_1.contains\,(e_2) \\ |e_1.startsWith\,(e_2)\,|e_1.endsWith\,(e_2)\,|e_1.isEmpty\,() \\ |c_1\&\&c_2|c_1||c_2|\neg c_i \end{aligned} \tag{2}$$

where $e_i \in SExp$ and $c_i \in BExp$

$$\mathscr{E}[[e_1.concat\,(e_2)]]\pi = concat\,(\mathscr{E}[[e_1]]\pi, \mathscr{E}[[e_2]]\pi) \tag{3}$$

$$\mathscr{E}[[e_1.substring\,(i,j)]]\pi = substring\,(\mathscr{E}[[e_1]]\pi, i, j) \tag{4}$$

$$\mathscr{E}[[e_1.insert\,(e_2,i)]]\pi = insert\,(\mathscr{E}[[e_1]]\pi, \mathscr{E}[[e_2]]\pi, i) \tag{5}$$

$$\mathscr{E}[[replace\,(e_1,e_2,e_3)]]\pi = replace\,(\mathscr{E}[[e_1]]\pi, \mathscr{E}[[e_2]]\pi, \mathscr{E}[[e_3]]\pi) \tag{6}$$

$$\mathscr{E}[[v]]\pi = \pi\,(v) \tag{7}$$

$$\mathscr{E}[[s]]\pi = DFA(s) \tag{8}$$

where $DFA(s)$ returns a DFA that accepts the string litteral s.

$$\mathscr{C}[[e_1.equals\,(e_2)]]\pi = \mathscr{A}[[e_1]](\mathscr{E}[[e_2]]\pi)\pi \sqcap \mathscr{A}[[e_2]](\mathscr{E}[[e_1]]\pi)\pi \tag{9}$$

$$\mathscr{C}[[e_1.contains\,(e_2)]]\pi = \left\langle \mathscr{A}[[e_1]](concat\,(\mathbf{1}, \mathscr{E}[[e_2]]\pi, \mathbf{1}))\pi \sqcap \mathscr{A}[[e_2]](substrings\,(\mathscr{E}[[e_1]]\pi))\pi \right\rangle \tag{10}$$

$$\mathscr{C}[[e_1.startsWith\,(e_2)]]\pi = \left\langle \mathscr{A}[[e_1]](concat\,(\mathscr{E}[[e_2]]\pi, \mathbf{1}))\pi \sqcap \mathscr{A}[[e_2]](prefixes\,(\mathscr{E}[[e_1]]\pi))\pi \right\rangle \tag{11}$$

$$\mathscr{C}[[e_1.endsWith\,(e_2)]]\pi = \left\langle \mathscr{A}[[e_1]](concat\,(\mathbf{1}, \mathscr{E}[[e_2]]\pi))\pi \sqcap \mathscr{A}[[e_2]](suffixes\,(\mathscr{E}[[e_1]]\pi))\pi \right\rangle \tag{12}$$

$$\mathscr{C}[[e_1.isEmpty\,()]]\pi = \mathscr{A}[[e_1]](\Lambda)\pi \tag{13}$$

$$\mathscr{C}[[\neg e_1.equals\,(e_2)]]\pi = \left\langle \mathscr{A}[[e_1]](complement\,(\mathscr{E}[[e_2]]\pi))\pi \;\sqcup\; \mathscr{A}[[e_2]](complement\,(\mathscr{E}[[e_1]]\pi))\pi \right\rangle \tag{14}$$

$$\mathscr{C}[[\neg e_1.contains\,(e_2)]]\pi = \left\langle \mathscr{A}[[e_1]](complement\,(concat\,(\mathbf{1}, \mathscr{E}[[e_2]]\pi, \mathbf{1})))\pi \sqcup \mathscr{A}[[e_2]](complement\,(substrings\,(\mathscr{E}[[e_1]]\pi)))\pi \right\rangle \tag{15}$$

$$\mathscr{C}[[\neg e_1.startsWith\,(e_2)]]\pi = \left\langle \mathscr{A}[[e_1]](complement\,(concat\,(\mathscr{E}[[e_2]]\pi, \mathbf{1})))\pi \sqcup \mathscr{A}[[e_2]](complement\,(prefixes\,(\mathscr{E}[[e_1]]\pi)))\pi \right\rangle \tag{16}$$

$$\mathscr{C}[[\neg e_1.endsWith\,(e_2)]]\pi = \left\langle \mathscr{A}[[e_1]](complement\,(concat\,(\mathbf{1}, \mathscr{E}[[e_2]]\pi)))\pi \sqcup \mathscr{A}[[e_2]](complement\,(suffixes\,(\mathscr{E}[[e_1]]\pi)))\pi \right\rangle \tag{17}$$

$$\mathscr{C}[[\neg e_1.isEmpty\,()]]\pi = \mathscr{A}[[e_1]](complement\,(\Lambda))\pi \tag{18}$$

$$\mathscr{C}[[c_1\&\&c_2]]\pi = \mathscr{C}[[c_1]]\pi \sqcap \mathscr{C}[[c_2]]\pi \tag{19}$$

$$\mathscr{C}[[c_1||c_2]]\pi = \mathscr{C}[[c_1]]\pi \sqcup \mathscr{C}[[c_2]]\pi \tag{20}$$

$$\mathscr{C}[[\neg(c_1\&\&c_2)]]\pi = \mathscr{C}[[\neg c_1||\neg c_2]]\pi \tag{21}$$

$$\mathscr{C}[[\neg(c_1||c_2)]]\pi = \mathscr{C}[[\neg c_1\&\&\neg c_2]]\pi \tag{22}$$

$$\mathscr{C}[[\neg(\neg c_1)]]\pi = \mathscr{C}[[c_1]]\pi \tag{23}$$

$$\mathscr{A}[[e_1.concat\,(e_2)]]a\;\pi = \left\langle \mathscr{A}[[e_1]](prefixes\,(concat\,(\mathbf{1}, \mathscr{E}[[e_2]]\pi) \sqcap\; a))\pi \sqcap \mathscr{A}[[e_2]](suffixes\,(concat\,(\mathscr{E}[[e_1]]\pi, \mathbf{1}) \sqcap\; a))\pi \right\rangle \tag{24}$$

$$\mathscr{A}[[e_1.insert\,(e_2, i)]]a\;\pi = \left\langle \mathscr{A}[[e_1]]\langle concat\,(substring\,(a, 0, i)\,, \mathbf{1}) \;\sqcup\; concat\,(\mathbf{1}, substring\,(a, i, length(a)))\rangle\pi \;\sqcap\; \mathscr{A}[[e_2]](substring\,(a, i, length(a_2)))\pi \right\rangle \tag{25}$$

where $length(a)$ is the maximal depth of $DFA\ a$

$$\mathscr{A}[[e_1.substring\,(i, j)]]a\;\pi = \mathscr{A}[[e_1]](concat(\mathbf{1}, (\mathscr{E}[[e_1.substring\,(i, j)]]\pi \sqcap\; a), \mathbf{1}))\pi \tag{26}$$

$$\mathscr{A}[[replace\,(e_1, e_2, e_3)]]a\;\pi = \mathscr{A}[[e_1]](\mathscr{E}[[replace\,(e_1, e_2, e_3)]]\pi \sqcap\; a)\pi \tag{27}$$

$$\mathscr{A}[[v]]a\;\pi = \pi[v \longrightarrow \pi\,(v) \sqcap a] \tag{28}$$

$$\mathscr{A}[[\ell]]a\;\pi = \begin{cases} \lambda v.\mathbf{0} & \text{if } \ell \notin L(a); \\ \pi & \text{Otherwise.} \end{cases} \tag{29}$$

While a branch-insensitive analysis propagates the identity function along the different branches of a boolean condition without applying any restrictions on the incoming data flow information, a branch-sensitive analysis restricts the incoming abstract store to satisfy a boolean expression which results in more precise results. Consider the following example of conditional restriction. $\mathscr{C}[[t.contains(s)]]\pi$ where $\pi = \{t \longleftarrow DFA($`"neutral(IEMI123)*"`$) = a_1, s \longleftarrow DFA($`"IEMI123"`$) = a_2\}$. π is the abstract store that associates t with the DFA that accepts the regular expression `"neutral(IEMI123)*"` and s with the DFA that accepts the string literal `"IEMI123"`.

$\mathscr{C}[[t.contains(s)]]\pi$
$= \langle \mathscr{A}[[t]]\ concat(\mathbf{1}, a_2, \mathbf{1})\pi \sqcap \mathscr{A}[[s]]\ substrings(a_1)\pi\ \rangle$

$= \pi[t \longleftarrow \pi(t) \sqcap concat(\mathbf{1}, a_2, \mathbf{1})]$
$\sqcap\pi[s \longleftarrow \pi(s) \sqcap substrings(a_1)]$

$= \{t \longleftarrow DFA($`"neutral(IEMI123)+"`$),$
$s \longleftarrow DFA($`"IEMI123"`$)\}$

Our analysis propagates $\{t \longleftarrow DFA($`"neutral(IEMI123)+"`$), s \longleftarrow DFA($`"IEMI123"`$)\}$ to the instructions of the true branch. The value propagated to the false branch is $\pi' = \mathscr{C}[[\neg t.contains(s)]]\ \pi$, the result of restricting the negation of the Boolean expression against the incoming abstract store π which is $\pi' = \{t \longleftarrow DFA($`"neutral"`$), s \longleftarrow DFA($`"IEMI123"`$)\}$.

Consider the following example of conditional restriction. $\mathscr{C}[[\neg t.contains(s)]]\pi$ where $\pi = \{t \longleftarrow DFA($`"neutral(IEMI123)+"`$) = a_3, s \longleftarrow DFA($`"IEMI123"`$) = a_2\}$. π is the abstract store that associates `t` with the DFA that accepts the regular expression `"neutral(IEMI123)+"` and `s` with the DFA that accepts the regular expression `"IEMI123"`.

$\mathscr{C}[[\neg t.contains(s)]]\pi$
$= \langle \mathscr{A}[[t]]\ complement(concat(\mathbf{1}, a_2, \mathbf{1}))\pi$
$\sqcup \mathscr{A}[[s]]\ complement(substrings(a_3))\pi\rangle$

$= \pi[t \longleftarrow \pi(t) \sqcap complement(concat(\mathbf{1}, a_2, \mathbf{1}))]$
$\sqcup\pi[s \longleftarrow \pi(s) \sqcap complement(substrings(a_3))]$

$= \{t \longleftarrow \mathbf{0}, s \longleftarrow \mathbf{0}\}$

Our analysis propagates $\lambda v.\mathbf{0}$ to the instructions of the true branch. The value propagated to the false branch is $\pi' = \mathscr{C}[[t.contains(s)]]\ \pi$, the result of restricting the negation of the boolean expression against the incoming abstract store π which is π since the regular expression of t always contains s.

VI. Abstract Operations for String analysis

Our string analysis builds on Vasco [17], a framework for implementing inter-procedural dataflow analysis of Java programs. We used a modified version of the framework. We implemented branch-sensitivity and used context-sensitivity based on data flow values, finite state machines in our case. Our string analysis is context-sensitive, which means that it is able to distinguish between the different function calls at the different call locations by tagging them with a calling context, a key that differentiates between the entries that correspond to the different call sites. The joining at merge points is performed by taking this context into account, and merging is performed for the data values by context rather than merging of all the data values regardless of which context they originated from.

String operations are given an abstract meaning as finite state machine transformers. Given that the lattice of DFAs has an infinite height, the widening technique proposed by [3] to safely approximate infinite string values generated in a loop is used to guarantee the convergence of the analysis. We use the `replace` operator proposed in [3]. Unlike JSA, the `replace` operator from [3] operates on finite state machines as well as string literals. Our implementation of the `substring` transducer builds on the algorithm presented in [18]. With DFA abstractions, abstract string concatenation is just the concatenation of two DFAs.

A. Abstract Substring Operation

The abstract `substring` operation from [18] performs a breadth-first traversal of the state nodes constituting the DFA and disregards the states that are reachable before the starting index and those that are reachable after the end index. A new initial and a new final state are added to the original machine. An epsilon transition is created from the initial state to every state reachable in start index steps. An epsilon transition is created from any state reachable in end index steps to the accepting state. If a final state is reached while looking for states reachable in start index steps, the initial state is made accepting and the empty string is accepted. If a final state is reached while looking for sates reachable in end index steps, the DFA accepts words whose length is less than the end index and that final state is kept as accepting in the resulting machine. The resulting state machine is then converted into an equivalent DFA.

The abstract `insert` operation functions in a similar manner and performs a breadth-first traversal of the state nodes of the DFA till it reaches those nodes that are reachable in start index steps. For each state reachable in start index steps an epsilon transition is added to the initial state of the DFA to be inserted and another epsilon transition is added form the final states of the DFA to that state. Only the final states of the host DFA are kept. The resulting state machine is then converted into an equivalent DFA.

B. Abstract Replace Operation

The abstract string `replace` from [3] $replace(M_1, M_2, M_3)$, consists of detecting every path in M_1 that accepts any word from $L(M_2)$ and replace that path by a copy of M_3. The algorithm operates in three stages. The first stage transforms M_1 into a machine M_1' that accepts words that are obtained from inserting pairs of separators into words from $L(M_1)$. The second step transforms M_2 into a machine M_2' that accepts the concatenation of words from M_2 and words that do not contains words from M_2 as substrings where the pairs of the same separators are used to delimit the two types of words. The last stage operates on the intersection of M_1' and M_2' and consists of detecting every path that accepts any word from M_2, which are the paths delimited by the separator, and replace that path by a copy of M_3.

To better understand the algorithm, consider the case where we want to replace every occurrence of numerals in the regular expression e_1 =`"[a-z][ a-z,0-9]*"` with the replace string `"***"`. The regular expression e_1 corresponds with the finite state machine presented in Fig.2. The output of the first transformation phase is M_1' shown in Fig.3. The separator

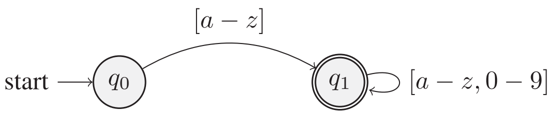

Fig. 2 Finite State Machine 1

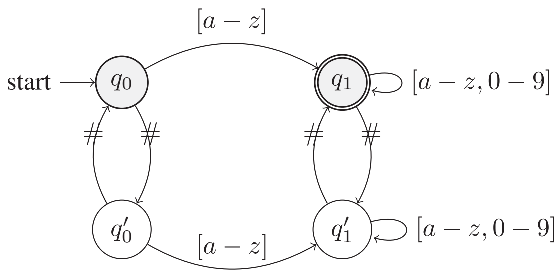

Fig. 3 Finite State Machine 2

character we used was #, the pound sign. The output of the second phase is shown in Fig.4. The machine accepts words where the numerals are delimited by the separator characters. The last stage detects the paths delimited by the separator character and replaces them with M2. The final machine returned by the replace operation is shown in Fig.5.

C. Widening

When the state domain satisfies the ascending chain property, the iterative approach for finding the least fixpoint solution of the data flow problem is guaranteed to converge in a finite number of steps. The least upper bound operator applies on merge points and a convergence test is used to check whether an additional iteration is still required. These two operators are enough in this case. In the case of a state domain with an infinite lattice, convergence is not guaranteed to be reached in a finite number of steps. It is where the widening operator is needed. Like the least upper bound a widening operation is an over-approximation of the original values. A widening operation results in a much broader over-approximation that results in a faster convergence. In order for the widening operation to be useful and not cause the analysis to lose information each time itâĂŹs applied, the over-approximation has to be constructed in a clever manner.

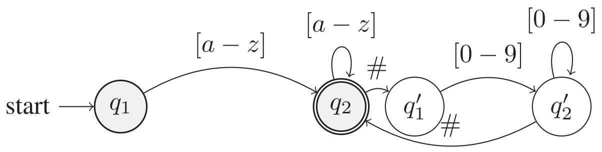

Fig. 4 Finite State Machine 3

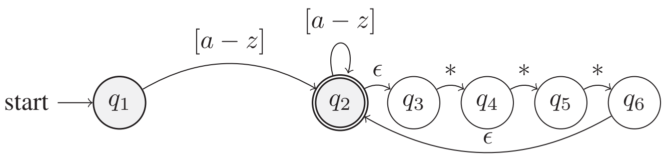

Fig. 5 Finite State Machine 4

We unraveled loops `k` times, applying the least upper bound operator on back edges till either convergence is achieved or the upper limit `k` is reached. If after the `kth` iteration the convergence is still not achieved, we apply the widening operation.

The widening operator $\triangledown$ was initially defined for arithmetic operations and later applied to DFAs in [3]. The widening of two finite state machines consists of defining an equivalence relation between the states of the two machines, where two states are equivalent iff for any word w from Σ^* both machines transit into an accepting state starting from these two states. The states of the resulting machine are the sets of equivalence classes. The initial state is the class that contains the initial states. The final states are equivalence classes that contain at least a final state and the transfer function is built from the initial transfer functions to transit from one equivalence class C_i to another equivalence class C_j on a given symbol iff the original transfer functions transit on any state from C_i to any state in C_j on that symbol.

$$L\left(M_1 \triangledown M_2\right) \sqsupseteq L\left(M_1\right) \sqcup L\left(M_2\right)$$

VII. Experimental Results

We combine our string model with existing static analyzers. The resulting product was used to improve the precision of the analyzer. It was also used to produce new analyses. We used our string model to improve the precision of taint analysis and to detect command injection attacks in Android programs.

We test our model on applications from different benchmarks [19], [20] and [21]. We also developed a set of custom benchmarks [22] specifically designed to showcase the benefits of an accurate model of string operations and their effect on the string variables of the program. Each of the benchmark apps embodies a common scenario of application development where operations such as `string.replace`, `string.concat`, `string.substring` and `string.insert` are used to manipulate the string variables. Conditional expressions such as `string.contains`, `string.startsWith`, `string.endsWith` are used to restrict the incoming abstract storeâĂŹs string variables so that it satisfies the conditional expression, potentially flagging it as unreachable when no such restriction was possible. The goal of the test was to accurately determine if the statement designated as sink was reachable, or if it belonged to an infeasible path, when the incoming abstract store at the entry of the sink is flagged as unreachable. A false positive occurs when the analysis assumes that the statement is reachable, when in fact it is not.

We ran our experiments on a Windows machine with a 2.6GHz Intel Core i5 processor and 8GB of RAM. The total number of expected unreachable sinks across all the benchmark apps was 31. Our analyzer reported exactly 31 infeasible paths. The average run-time the analyzer took to analyze a benchmark was 837.23ms.

Our experimental results are summarized in Table 1. The column under Avg runtime shows the average run-time in

TABLE I
EXPERIMENTAL RESULTS

Benchmark (# of apps, LOC)	Avg runtime(ms)	% Time op: op	% Time op: time	# IS	# FP	Memory (MB)
TASA (64, 1500)	837.23	replace	9.69%	31	0	75
		substring	1%			
		insert	1%			
		contains	35.91%			
		startsWith	31.05%			
		endsWith	21.72%			

milliseconds that the string analyzer took to complete a benchmark app. The columns under % Time op shows the percentage of the ration of the analysis time of that string operation to the total string analysis time. The column under #IS shows the total number of unreachable sinks reported by the tool. The column under #FP record the number of false positives reported by the tool. The column under Memory records the memory consumption reported by the tool in MB.

VIII. CONCLUSION

Static analysis of library method invocation can be impossible to achieve at times because some libraries are not entirely written in Java. As such, the need exists for an intelligent model that approximates the behavior of library method invocations to improve the precision of analysis. We build a model of string operations based on DFAs and we use it as a building block in other analyses. The information made available by the product of the two analyses was useful at improving the precision of the analyzer. Our model proves to be very efficient at detecting when a given statement, enclosed in an unrealizable conditional expression, was unreachable; specifically when the abstract store could not be restricted in a way that satisfies the condition.

REFERENCES

[1] "Open web application security project.." Available at https://www.owasp.org.
[2] A. S. Christensen, A. Møller, and M. I. Schwartzbach, "Precise analysis of string expressions," in *International Static Analysis Symposium*, pp. 1–18, Springer, 2003.
[3] F. Yu, T. Bultan, M. Cova, and O. H. Ibarra, "Symbolic string verification: An automata-based approach," in *International SPIN Workshop on Model Checking of Software*, pp. 306–324, Springer, 2008.
[4] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps," *SIGPLAN Not.*, vol. 49, pp. 259–269, June 2014.
[5] D. Li, Y. Lyu, M. Wan, and W. G. Halfond, "String analysis for java and android applications," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pp. 661–672, ACM, 2015.
[6] D. Octeau, P. McDaniel, S. Jha, A. Bartel, E. Bodden, J. Klein, and Y. Le Traon, "Effective inter-component communication mapping in android: An essential step towards holistic security analysis," in *Presented as part of the 22nd {USENIX} Security Symposium ({USENIX} Security 13)*, pp. 543–558, 2013.
[7] E. Bodden, "Inter-procedural data-flow analysis with ifds/ide and soot," in *Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program analysis*, pp. 3–8, ACM, 2012.
[8] A. Einarsson and J. D. Nielsen, "A survivorâĂŹs guide to java program analysis with soot," *BRICS, Department of Computer Science, University of Aarhus, Denmark*, p. 17, 2008.
[9] C. Fritz, S. Arzt, S. Rasthofer, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "Highly precise taint analysis for android applications," 2013.
[10] R. Amadini, A. Jordan, G. Gange, F. Gauthier, P. Schachte, H. Søndergaard, P. J. Stuckey, and C. Zhang, "Combining string abstract domains for javascript analysis: an evaluation," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pp. 41–57, Springer, 2017.
[11] G. Costantini, P. Ferrara, and A. Cortesi, "A suite of abstract domains for static analysis of string values," *Software: Practice and Experience*, vol. 45, no. 2, pp. 245–287, 2015.
[12] M. Madsen and E. Andreasen, "String analysis for dynamic field access," in *International Conference on Compiler Construction*, pp. 197–217, Springer, 2014.
[13] S. H. Jensen, A. Møller, and P. Thiemann, "Type analysis for javascript," in *International Static Analysis Symposium*, pp. 238–255, Springer, 2009.
[14] V. Kashyap, K. Dewey, E. A. Kuefner, J. Wagner, K. Gibbons, J. Sarracino, B. Wiedermann, and B. Hardekopf, "Jsai: a static analysis platform for javascript," in *Proceedings of the 22nd ACM SIGSOFT international symposium on Foundations of Software Engineering*, pp. 121–132, ACM, 2014.
[15] H. Lee, S. Won, J. Jin, J. Cho, and S. Ryu, "Safe: Formal specification and implementation of a scalable analysis framework for ecmascript," in *FOOL 2012: 19th International Workshop on Foundations of Object-Oriented Languages*, p. 96, Citeseer, 2012.
[16] F. Yu, T. Bultan, and O. H. Ibarra, "Relational string verification using multi-track automata," *International Journal of Foundations of Computer Science*, vol. 22, no. 08, pp. 1909–1924, 2011.
[17] R. Padhye and U. P. Khedker, "Interprocedural data flow analysis in soot using value contexts," in *Proceedings of the 2nd ACM SIGPLAN International Workshop on State Of the Art in Java Program analysis*, pp. 31–36, ACM, 2013.
[18] N. Almashfi, L. Lu, K. Picker, and C. Maldonado, "Precise string analysis for javascript programs using automata," in *Proceedings of the 2019 8th International Conference on Software and Computer Applications*, pp. 159–166, ACM, 2019.
[19] "Droidbench benchmark suite.," 2020. Available at https://github.com/secure-software-engineering/DroidBench.
[20] "Icc-bench benchmark suite.," 2020. Available at https://github.com/fgwei/ICC-Bench.
[21] L. Qiu, Y. Wang, and J. Rubin, "Analyzing the analyzers: Flowdroid/iccta, amandroid, and droidsafe," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 176–186, ACM, 2018.
[22] "Taint analysis of strings with automatons.," 2020. Available at https://drive.google.com/file/d/1RmxuFvk6TCCFxSuUuUss9cUVKI3zHOwV/view?usp=sharing.