International Journal of Information, Control and Computer Sciences
ISSN: 2517-9942
Vol:14, No:12, 2020

Improving Taint Analysis of Android Applications Using Finite State Machines

Assad Maalouf, Lunjin Lu, James Lynott

Abstract—We present a taint analysis that can automatically detect when string operations result in a string that is free of taints, where all the tainted patterns have been removed. This is an improvement on the conservative behavior of previous taint analyzers, where a string operation on a tainted string always leads to a tainted string unless the operation is manually marked as a sanitizer. The taint analysis is built on top of a string analysis that uses finite state automata to approximate the sets of values that string variables can take during the execution of a program. The proposed approach has been implemented as an extension of FlowDroid and experimental results show that the resulting taint analyzer is much more precise than the original FlowDroid.

Keywords—Android, static analysis, string analysis, taint analysis.

I. Introduction

ANY path in the code where sensitive information, such as contact information, location coordinates or SMS messages, is broadcast though SMS or email or written to a network socket can constitute a serious breach in security policy. Static taint analysis tracks suspicious data flows, paths in the code where a piece of private information leaks through a public sink. Previous static taint analyzers for Android applications such as [1]- [2] generate a large number of false positives. A false positive occurs when the analysis reports a potential leak when in reality, the path does not constitute an actual violation of the security policy. Those analyzers are not precise enough because they use coarse grained labeling that treats a string operation as either a taint sanitizer or a taint generator or a taint propagator regardless the context in which it is applied. This classification cannot handle the case where the same operation can behave as a taint sanitizer in one context and as a taint generator / propagator in another context. For instance, a string replacement operation can behave as a taint sanitizer or as a taint generator / propagator depending on the context of invocation. A call to replace, whose effect is to detect all the occurrences of a tainted pattern and to replace it with a sanitized string results in a string free of taint. A call to replace that injects a taint in some previously untainted string behaves as taint generator. The same applies for the substring operation. Substring can return parts of the string that do not contain any taint. Previous approaches treat the return value of an operation as tainted if it involves a tainted argument. This is safe but it cannot detect the case where the operation acts as a taint sanitizer. This paper presents a static taint analysis that is able to determine if a string operation is used in a given context as a taint generator, a taint propagator, or a taint sanitizer. This is achieved by associating each string expression with a set of possible taint sources. The taint sources are the malicious patterns we want to guarantee they do not occur in the final string.

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

The main contributions of this paper are as follows:

1) Design and implement a string analysis of Android applications using finite automata to approximate the sets of values that string variables might take during execution.
2) Make use of these approximations to improve the precision of taint analysis of string expressions and to discover unreachable code.
3) Implement a prototype analyzer, $FlowDroid_{STR}$, on top of FlowDroid and evaluate it on multiple applications that handle strings. We also evaluate $FlowDroid_{STR}$ on a set of custom benchmarks, TASA [3], specifically designed for the purpose of this study.

The rest of the paper is organized as follows. Section II presents some motivating examples. Section III discusses the related work on taint analysis. Section IV presents the string analysis of Android programs. Section V describes taint analysis using string analysis to improve taint propagation. Section VI presents experimental results. Section VII concludes with some notes on future work.

II. Motivating Examples

As a motivating example, consider this code to which line numbers have been added for purpose of easy exposition.

```
1 String s = TelephonyManager.getDeviceId();
2 String t = "neutral";
3 t = t.concat(s);
4 (new ConnectionManager())
      .publish(t.replace(s, "neutral"));
```

Fig. 1 Code Snippet 1

The instruction at line 1 reads a piece of sensitive information and assigns it to variable s. After the first instruction s is tainted. The instruction at line 4 broadcasts the result of an operation that involves a tainted argument. Previous taint analyzers treat it as a leak. A closer inspection of the replace operation allows us to conclude that every occurrence of s is replaced with "neutral" and the result does not carry any sensitive information. The instruction does not cause an actual leak. $FlowDroid_{STR}$ tracks the taint s as it gets injected into the string t. It keeps track of all the atomic taints that have been used in the computation of the string

expression t. It precisely approximates *replace* and *substring* operations and concludes that the string that can occur at line 4 does not contain taint.

In the code below, at the entry of the instruction at line 7 the string variable t can either carry the result of the assignment at line 3 or the result of the assignment at line 4. The conditional expression at line 6 constraints the values of t and the only possible value of t at line 7 can come from the assignment at line 4. This assignment does not carry any sensitive information and the publish does not constitute an actual leak. $FlowDroid_{STR}$ is branch sensitive and is capable of accurately modeling conditional expressions on strings and uses them to improve the precision of taint analysis.

```
1 String s = TelephonyManager.getDeviceId();
2 String t = "neutral";
3 if (Math.random() ≤ 0.5) t = t.concat(s);
4 else t = t.concat(t);
5 ConnectionManager cm = new ConnectionManager();
6 if (t.endsWith("neutral"))
7 cm.publish(t);
```

Fig. 2 Code Snippet 2

III. Related Work

There has been much research into taint analysis of Android applications, because of its importance in detecting security violations. FlowDroid [4], [5] is a flow-sensitive, context-sensitive taint flow analyzer of Android components, it is able to detect flow from a piece of private information into a public sink. FlowDroid does not analyze the interactions among the different components of a given application nor does it rely on real-time analysis of the code to detect the lifecycle. The tool constructs a dummy main method to simulate the lifecycle. This approach builds on predefined schemes and cannot model every combination of lifecycle ordering. FlowDroid is not value-sensitive and does not approximate the runtime values of the variables in the program. It relies on Soot's Constant Propagation modules to optimize its call graph generation and detect unreachable, dead code. For library method invocation, FlowDroid adopts a conservative and safe approach. It treats library method invocation as taint propagators, safely tainting the return value resulting from a manipulation of a tainted argument.

IccTA [6] is an extension of FlowDroid for inter-component communication (ICC for short). Android components communicate using intents. An intent object can target a component that resides within the current app or outside the current app and can alter the control as well as the data flow within the application code. IccTA builds on the results FlowDroid produces for a single component, detects ICC links and replaces those links with the target component of the link. IccTA is flow, object and field-sensitive but not value-sensitive. A value-sensitive analyzer approximates runtime values and makes use of these approximations to refine its main analysis, one such refinement is detecting infeasible paths in the code, which results in a more precise analysis and improves the overall performance of analysis [1].

DidFail [7] is another extension of FlowDroid for inter-component communication. DidFail builds on Epicc and FlowDroid to perform a constraint-based taint analysis. Constraints are added using the intra-component tainted paths discovered by FlowDroid and the ICC edges to target components computed using Epicc for intent analysis. These targets can reside in more than one application. The solution to these constraints determines the set of taints that might reach a sink across multiple applications.

Gator [8] is a general-purpose information flow analyzer. Gator provides its users with a custom data structure called a callback control flow graph (CCFG for short) and allows them to discover ill-formed paths and tainted paths. An example of an ill-formed path is a path where a resource is acquired and never released or a path where some private information leaks through a public sink.

Amandroid [2] is a tool that handles the inter-component control and data flow between different components, even when they reside in more than one app. Amandroid takes into account ICC intents and how they affect the control and flow of data facts. Amandroid is flow and context-sensitive but does not model some important Android constructs such as startActivityForResult, nor does it model all components of an Android application. Amandroid builds a precise model to compute the effect of critical API calls, such as the calls that handle intents. Amandroid adopts a conservative approach for all other library method invocation and assumes that a method can return any object visible in the scope of the function, reachable from the method's parameters and is compatible with the method's return type.

HornDroid [1] is another tool for static analysis of Android programs. HornDroid uses abstract interpretation and builds a formal abstract model of Horn clauses from the concrete semantics of the Android operations. The application is transformed into a set of logical rules, and the security properties become logical queries that are solved using off-the-shelf SMT solvers. HornDroid focuses its analysis on the semantics of Android specific constructs. HornDroid implements a simple string analysis and uses it to resolve reflective calls and intent manipulation. HornDroid adopts a conservative model for library method invocation and treats the return value of a call as tainted whenever one of the arguments of the call is tainted.

Epicc [9] analyses the flow of information between different components of Android applications. Epicc reduces the problem of inter-component communication of Android applications to an instance of an IDE [10] problem. Epicc identifies malicious behavior resulting from the cooperation of multiple components working side by side to accomplish a malicious action. Application collusion is an example of such an attack. Android collusion involves two or more applications. Although each application individually does not exhibit any malicious behavior, their aggregate behavior can result in a dangerous attack. 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. Android components can exchange data

through the form of a key-value pair within intent objects. An intent object intercepted by a malicious component can result in information leakage, and malicious intents can compromise the security of the system. In Epicc, distributive environment transformers model the effects of API method calls on the intents and the data they carry. The problem is solved using Heroes and Soot.

While FlowDroid [5] and IccTA [6] are value insensitive and rely completely on Soot modules to optimize their call graph generation and to eliminate unreachable code. Gator [8] , Amandroid [2], Epicc [9] and HornDroid [1] are value sensitive and benefit from a simple constant propagation analysis. This allows them 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.

IV. String Analysis

A. Abstract Domain

There has been much work on string analysis of programs in Java / Android and other programming languages [9], [11]- [12]. The abstract domains proposed for stings analysis range from constant propagation [13], to multitrack finite automata [12], providing different trade-offs between precision and cost of analysis. We use the domain of deterministic finite state automata DFA over the alphabet of Unicode Σ to approximate the sets of strings that variables might take during program execution. The abstract domain captures string properties sufficiently precisely and it admits sufficiently efficient implementation. The string analysis is built on Vasco [14] - a framework for implementing inter-procedural dataflow analysis of Java programs. The framework is modified to support value-based branch-sensitivity and context-sensitivity and is then instantiated with an abstract domain for string analysis. The abstract domain is equipped with several abstract string operations that approximate concrete string operations. Each concrete string operation is approximated by an abstract string operation.

B. Abstract Operations

Concrete operations on strings fall into two categories: those that return string values and those that return boolean values. We first present abstract operations that approximate those concrete operations that return string values. These abstract operations are finite automaton transformers and are implemented by extending JSA's [15] library of transducers with a more elaborate form of string operations on finite automata. With DFA abstractions, abstract string concatenation is just the concatenation of two DFAs. A StringBuilder is also represented with a DFA and the abstract append operation is modeled using the concatenation of DFAs. In what follows let $L(M)$ be the language of DFA M, the set of words from Σ^* that are accepted by the automaton M.

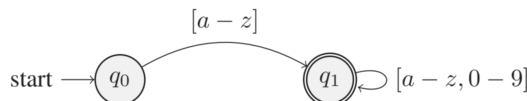

Fig. 3 Finite State Machine 1

1) Abstract Substring operation: The abstract substring operation is based on an algorithm presented in [16]. It 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 automaton. 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 automaton. The resulting automaton is then converted into an equivalent DFA.

2) Abstract Replace Operation: The abstract string replace operation replace(M_1,M_2, M_3) from [17] is used in string analysis. It 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 an automaton 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 an automaton 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. Unlike the replace operator provided by JSA, which is only able to handle cases where operands are string literals, the replace operator in [17] operates on finite automata. This greatly improves the precision of analysis.

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.3. The output of the first transformation phase is M_1' shown in Fig.4. The separator character we used was `#`, the pound sign. The output of the second phase is shown in Fig.5. 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.6.

3) Widening: Given that the lattice of DFAs has an infinite height, the widening operator from [17] to safely approximate infinite set of string values generated in a loop is used to

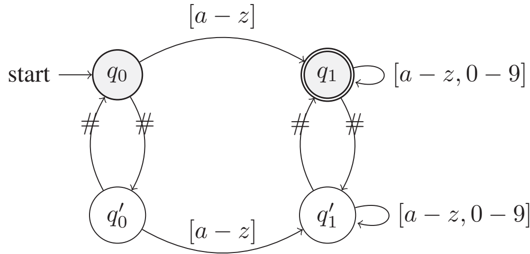

Fig. 4 Finite State Machine 2

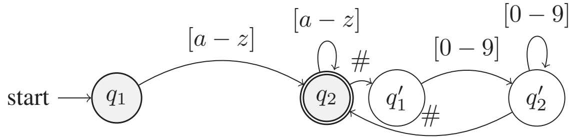

Fig. 5 Finite State Machine 3

guarantee the convergence of the analysis. We unroll loops up to some predefined limit. If convergence is not achieved within the limit, we apply the widening operator.

The widening operator ∇ was initially defined for arithmetic operations and later applied to DFAs in [17]. The widening of two finite automata consists of defining an equivalence relation between the states of the two automata, where two states are equivalent iff for any word w from Σ^* both automata transit into an accepting state starting from these two states. The states of the resulting automaton 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.

C. Branch Sensitivity

We now present abstract operations that approximate those concrete operations that return boolean values. They are abstract store transformers. An abstract store is a mapping from string variables to finite automata. Let e_1 and e_2 be two string expressions and σ_{in} be the input abstract store. A boolean expression of the form $e_1.op(e_2)$ where op is a comparison operator transforms the input abstract store σ_{in} to an output abstract store σ_{out} as follows. The strings expressions e_1 and e_2 are abstractly evaluated in σ_{in} to obtain finite automata a_1 and a_2 respectively. Then an abstract store σ_1 is obtained from e_1 and a_2 and another abstract store σ_2 is obtained from e_2 and a_1. Finally, the output abstract store σ_{out} is obtained as the meet of σ_{in} and σ_1 and σ_2 where the meet operation on abstract stores is the point-wise extension of the intersection operation on finite automata. For different comparison operator op, σ_1 and σ_2 are obtained differently as follows.

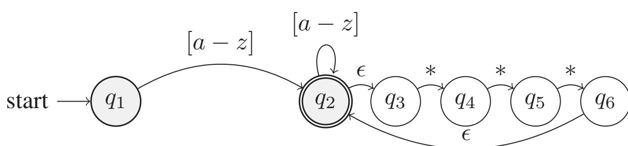

Fig. 6 Finite State Machine 4

1) Abstract Equals operation: The sets of the values of the string variables in e_1 are approximated by finite automata, such that e_1 is a string accepted by a_2, giving rise to an abstract store σ_1, and the sets of values of the string variables in e_2 are approximated by finite automata, such that e_2 is a string accepted by a_1, giving rise to an abstract store σ_2.

2) Abstract Contains operation: A finite automaton a'_2 is constructed such that a'_2 accepts a string w iff a word in $L(a_2)$ occurs as a substring in w, and a finite automaton a'_1 is constructed such that a'_1 accepts a string w iff w is a substring of some string in $L(a_1)$. The sets of the values of the string variables in e_1 are approximated by finite automata, such that e_1 is a string accepted by a'_2, giving rise to an abstract store σ_1, and the sets of values of the string variables in e_2 are approximated by finite automata, such that e_2 is a string accepted by a'_1, giving rise to an abstract store σ_2.

3) Abstract StartsWith operation: A finite automaton a'_2 is constructed from a_2 such that a'_2 accepts a string w iff w contains a string in $L(a_2)$ as a prefix, and a finite automaton a'_1 is constructed such that a'_1 accepts a string w iff w is a prefix of a string in $L(a_1)$. Then σ_1 and σ_2 are obtained as in the abstract contains operation.

4) Abstract EndsWith operation: A finite automaton a'_2 is constructed such that a'_2 accepts a string w iff w contains a string in $L(a_2)$ as a suffix, and a finite automaton a'_1 is constructed such that a'_1 accepts a string w iff w is a suffix of a string in $L(a_1)$. Then σ_1 and σ_2 are obtained as in the abstract contains operation.

V. Taint Analysis of String Expressions

We refine the semantics of taint analysis and we make them aware of the contextual effect of string operations on taint propagation. We use $source()$ in place of any instruction that has the potential of returning a taint, i.e., a piece of sensitive information, in the program. We refer to the left hand side of an assignment $v = source()$, as an atomic taint. Any value computed in terms of an atomic taint is a potentially tained value. A taint analyzer typically tracks a taint as it flows to reach a sink. A sink is any instruction that has the potential of leaking sensitive information, resulting in disclosure of critical data. $FlowDroid_{STR}$ keeps track of a list of variables that have potentially tainted values. These variables are called abstract taints. $FlowDroid_{STR}$ abstractly evaluates a tainted string expression into a pair consisting of a DFA and a set of atomic taints. The DFA approximates the set of possible values of the string expression and the set contains the atomic taints that were used to compute the value of the string expression. The set of the atomic taints keeps track of the sources of the taints that got injected into the string expression through string operations *concat*, *append*, *replace* and *substring*. Let $atoms(v)$ be the set of atomic taints associated with the string variable v.

The Android instructions that involve string operations are analyzed as follows.

Case 0: $v = source()$ where v is a string variable and the right hand side is a source of taint. The variable v is added to the list of abstract taints. The set of atomic taints associated with v is now $\{v\}$. $atoms(v) = \{v\}$.

Case 1: $v = v1.concat(v2)$ updates the set of atomic taints of v to become $atoms(v) = atoms(v1) \cup atoms(v2)$

Case 2: $v1.append(v2)$ updates the set of atomic taints of $v1$ to become $atoms(v1) = atoms(v1) \bigcup atoms(v2)$

Case 3: $v = v1.toString()$ updates the set of atomic taints of v to become $atoms(v) = atoms(v1)$

Case 4: $v1 = v2$ where $v1$ and $v2$ are string variables updates the set of atomic taints of $v1$ to become $atoms(v1) = atoms(v2)$

Case 5: $v = v1.substring(i, j)$ updates the set of atomic taints of v to become $atoms(v) = atoms(v1)$

Case 6: $v = v1.replace(v2, v3)$ updates the set of atomic taints of v to become $atoms(v) = atoms(v1) \bigcup atoms(v3)$

Case 7: Any broadcast operation that involves a string expression `e`, $FlowDroid_{STR}$ checks if the value of the string expression contains any atomic taints as substring. This is done as follows. For each atomic taint from the set of atomic taints associated with `e` and associated with DFA A, $FlowDroid_{STR}$ checks if $\mathbf{1}.A.\mathbf{1}$ intersects with E - the DFA associated with the string expression of `e`, where $\mathbf{1}$ is a constant DFA such that $L(\mathbf{1}) = \Sigma^*$. If they intersect, the value of `e` is potentially tainted, otherwise the automaton E does not accept any string that can contain a tainted pattern as a substring. In this case the value being broadcast is definitely not tainted.

An atomic taint associates one or more malicious patterns with a string expression; A malicious pattern is a regular expression that encapsulates the taint. The last check consists of detecting if those malicious patterns can still occur in the final string. Reformatting a malicious pattern or attempting to fragment the atomic pattern are detected by the analysis; An atomic taint is always associated with the different patterns in which it can occur and the substring abstract operation is redefined to always return the entire pattern when its start and end indexes happen to fall within the boundaries of the atomic pattern.

State of the art static taint analyzers of Android programs rely on general, less precise rules to cover most of the cases of taint propagation resulting from library function calls. FlowDroid implements the semantics of taint propagation as defined in [4]. FlowDroid, by default, treats library calls as taint propagators, unless they are manually classified as generators or sanitizers. In the case of an assignment, FlowDroid will consider the left-hand side of the assignment to be tainted if any of the operands in the right hand side is tainted regardless of the operation being performed on these operands. An invocation of taint generator with a tainted argument taints the base object in the case of an instance method call, it also taints the return value, and the left-hand side of the assignment is labeled as tainted as a result of the call. The basic idea of a taint generator / propagator is that a manipulation of an object that stores tainted data can potentially return tainted data and hence, the return value of that method call is labeled as tainted.

$FlowDroid_{STR}$ implements our approach to taint propagation, a refinement of the semantics of [4]. $FlowDroid_{STR}$ benefits from a more precise modeling of string operations based on finite state automata. It implements the abstract replace operation from [17], and the abstract substring operation from [16]. $FlowDroid_{STR}$ also models the effect of conditional expressions on automata, in order to detect the effect of contextual taint sanitizers as described earlier.

Consider the following example where results of string and taint analyses are comments.

```
1  //String: {s← 1 , t← 1, u← 1}
2  //Taint:  {}
3  String s = TelephonyManager.getDeviceId();
4  //String: {s← DFA("IEMI1") , t← 1, u← 1}
5  //Taint:  {(s,{s})}
6  String u = TelephonyManager.getSubscriberId();
7  //String: {s← DFA("IEMI1") , t← 1, u←
       DFA("IMSI1")}
8  //Taint:  {(s,{s}),(u,{u})}
9  String t = "neutral";
10 //String: {s← DFA("IEMI1") , t←
       DFA("neutral"),u← DFA("IMSI1")}
11 //Taint:  {(s,{s}),(u,{u})}
12 if (Math.random() ≤ 0.5){
13 t = t.concat(s).concat(u);
14 //String: {s← DFA("IEMI1") , t←
       DFA("neutralIEMI1IMSI1"),u← DFA("IMSI1")}
15 //Taint:  {(s,{s}),(u,{u}),(t,{s,u})}
16 else {
17 t = "litteral";
18 //String: {s← DFA("IEMI1") , t←
       DFA("litteral"),u← DFA("IMSI1")}
19 //Taint:  {(s,{s}),(u,{u})}
20 }
21 //String: {s← DFA("IEMI1") , t←
       DFA("neutralIEMI1IMSI1|litteral"),u←
       DFA("IMSI1")}
22 //Taint:  {(s,{s}),(u,{u}),(t,{s,u})}
23 ConnectionManager cm = new ConnectionManager();
24 if (!t.startsWith("neutral")){
25 //String: {s← DFA("IEMI1") , t←
       DFA("litteral"),u← DFA("IMSI1")}
26 //Taint:  {(s,{s}),(u,{u}),(t,{s,u})}
27 cm.publish(t);
28 }
```

Fig. 7 Code Snippet 3

An android app that has been granted the permission to READ_PHONE_STATE can access the unique identifiers of an Android device by invoking operations such as TelephonyManager.getSubscriberId(), TelephonyManager.getMeid(), TelephonyManager.getDeviceId(). The instruction TelephonyManager.getDeviceId() for example, returns the IMEI ID of the phone, a concrete string that uniquely identifies the phone. After the instruction at line 3 the string analysis associates `s` with the automaton that accepts the unique ID of the device. After the instruction at line 6 the string analysis associates `u` with the automaton that accepts the unique subscriber ID of the device owner. At the entry of the instruction at line 23, `t` can be the result of the assignment at line 13 or the result of the

assignment at line 17. The conditional expression at line 24 restricts t to only those strings that do not start with literal "neutral". TelephonyManager.getDeviceId() and TelephonyManager.getSubscriberId() are taint sources. After the instruction at line 3, s is added to the list of taints and $atoms(s) = \{s\}$. After the instruction at line 6, u is added to the list of taints and $atoms(u) = \{u\}$. After the instruction at line 13, t is tainted and the $atoms(t) = \{s, u\}$. Before the publish instruction at line 27, t is associated with DFA("litteral") and $atoms(t) = \{s, u\}$. Since neither s nor u can occur as a substring of words of t, the publish instruction does not leak any sensitive information.

Consider the following example with a loop.

```
1  //String: {s←— 1 , t←— 1}
2  //Taint:  {}
3  String s = TelephonyManager.getDeviceId();
4  //String: {s←— DFA("IEMI1") , t←— 1}
5  //Taint:  {(s,atoms(s)={s})}
6  String t =  "neutral";
7  //String: {s←— DFA("IEMI1") , t←— DFA("neutral")}
8  //Taint:  {(s,atoms(s)={s})}
9  while(Math.random() ≤ 0.5)  {
10 //String: {s←— DFA("IEMI1") , t←—
       DFA("neutral(IEMI1)*")}
11 //Taint:  {(s,{s})}
12 t=t.concat(s);
13 //String: {s←— DFA("IEMI1") , t←—
       DFA("neutral(IEMI1)+")}
14 //Taint:  {(s,{s}),(t,{s})}
15 }
16 //String: {s←— DFA("IEMI1") , t←—
       DFA("neutral(IEMI1)*")}
17 //Taint:  {(s,{s}),(t,{s})}
18 ConnectionManager cm = new ConnectionManager();
19 if (!t.contains(s)) {
20 //String:{s←— DFA("IEMI1") , t←— DFA("neutral")}
21 //Taint: {(s,{s}),(t,{s})}
22 cm.publish(t);
23 }
```

Fig. 8 Code Snippet 4

After the instruction at line 3, s is associated with the automaton that accepts the unique device ID. At the exit of the while loop, t can have 0 or more s appended to it and the string analysis approximates t with the regular expression "neutral(IEMI1)*". The conditional expression at line 19 restricts t to only those that do not contain s. The variable t is associated with the regular expression "neutral". The results of the string analysis are made available to the taint analyzer. After the first instruction s is tainted and $atoms(s) = \{s\}$. After the instruction at line 12, t the result of an operation on a tainted argument, it is added to the list of abstract taints and $atoms(t) = \{s\}$. None of the atoms of t can occur as a substring in a word accepted by the automaton DFA("neutral") and therefore the publish instruction is not leaky.

VI. Experimental Results

For the purpose of evaluating our work and showcasing the benefits of coupling taint analysis with an accurate string analysis we evaluate our tool on applications from DroidBench [18], ICC-Bench [19] and UBCBench [20] benchmark suites. We also extend the set of existing benchmarks with our own benchmark applications, TASA [3]. We develop a set of 64 benchmark applications, each representing a specific scenario of manipulation of a tainted string and transformation of the taint with string operations. We name our benchmarks TASA, short for taint analysis of strings with automata. Each of the benchmark apps embodies a common scenario of application development where operations such as string.replace, string.append and string.concat were used to inject a taint into a previously untainted string. Operations such as string.replace was used to sanitize the tainted part of a string. Operations such as string.substring were used to retrieve parts of a string that fell into the tainted part of the string or fell into the sanitized part of the string. Conditional expressions such as string.contains, string.startsWith, string.endsWith were used to restrict the incoming strings to a given condition and potentially eliminating tainted patterns from the string so that it satisfies the conditional expression. We also evaluate $FlowDroid_{STR}$ on random apps from the Google Play Store. The problem with random apps resides in the difficulty in deciding whether a leak reported by the tool constitutes an actual breach in the security policy. We report our evaluation of $FlowDroid_{STR}$ on TASA because the results are not ambiguous and we make our detailed results available at [3].

We ran our experiments on a Windows machine with 2.6GHz Intel Core i5 processor and 8GB of RAM. Our experimental results are summarized in Table I. The two columns under #FP record the number of false positives reported by FlowDroid and $FlowDroid_{STR}$ respectively. The three columns under Time record the analysis times reported by FlowDroid, $FlowDroid_{STR}$ and the ratio of the analysis time reported $FlowDroid_{STR}$ to the analysis time reported by FlowDroid. The two columns under Memory record the memory consumption reported by FlowDroid and $FlowDroid_{STR}$ respectively. The number of expected leaks for a given app was determined by manually inspecting the flows of the app. The total number of expected leaks across all the benchmark apps is 31. $FlowDroid_{STR}$ reports exactly 31 leaks. FlowDroid reports a total of 63 leaks out of which 32 were false positives. The average run-time $FlowDroid_{STR}$ took to analyze a benchmark was 5,769.92ms. The average run-time FlowDroid took to analyze a benchmark was 4,563.84 ms.

The following observations were made:

1) The precision of the tool is computed as being the percentage of the ratio of the number of false positives to the total number of reported flows. $FlowDroid_{STR}$ is twice as precise as FlowDroid on TASA.
2) FlowDroid and $FlowDroid_{STR}$ consumed approximately the same memory.
3) $FlowDroid_{STR}$ is 27% slower than FlowDroid.

The experimental results show that a precise string analysis is very beneficial when it comes to accurately deciding the effect of string operations on taint sanitization. They also show that the overhead of coupling taint analysis with a string analysis based on finite automata is acceptable given that static taint analysis is often performed offline.

While randomly selected mobile apps from the Google Play

TABLE I
EXPERIMENTAL RESULTS

Benchmark (# of apps, LOC)	#FP		Time(ms)			Memory (MB)	
	F	F_{STR}	F	F_{STR}	Time %	F	F_{STR}
TASA (64 apps, 15000)	32	0	4563.84	5769.92	126.42%	73	75

Store may not contain malicious flows, accidental leaks are common in real world application. Deciding whether a leak detected by the analyzer constitutes an actual security breach may require additional considerations to determine if the setup of the environment in which the app runs prevents the leak from constituting an actual security vulnerability. In addition to evaluating $FlowDroid_{STR}$ on multiple randomly selected apps from the Google Play Store we evaluate our tool on InsecureBank [21]. InsecureBank is an Android application used for penetration testing and developed in the purpose of simulating a real-world mobile application with embedded vulnerabilities similar to those found in real-world scenarios. We inspect the source code and manually identify the different leak. The version of the code contained 6 leaks. FlowDroid and $FlowDroid_{STR}$ successfully find all of them in about 30s.

VII. Conclusion and Future Work

We have presented a precise taint analysis that keeps track of sources of taints in string expressions and makes use of a string analysis based on finite automata. The analysis is implemented as an extension of Flowdroid and results in a more precise taint analyzer $FlowDroid_{STR}$. $FlowDroid_{STR}$ is able to improve on the number of false positives of Flowdroid's taint propagation by implementing a precise flow sensitive string analysis. It is able to detect infeasible paths in the code and detect the taints that do not occur as a result of the source or the sink being unreachable. We plan to experiment with other forms of numerical and string analysis and the effects they might have when coupled with a state of the art taint analyzer like FlowDroid. Future work would investigate the tradeoffs of precision to cost these other analyses might incur on the original analysis.

References

[1] S. Calzavara, I. Grishchenko, and M. Maffei, "Horndroid: Practical and sound static analysis of android applications by smt solving," in *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*, pp. 47–62, IEEE, 2016.

[2] F. Wei, S. Roy, X. Ou, *et al.*, "Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pp. 1329–1341, ACM, 2014.

[3] "Taint analysis of strings with automatons.," 2020. Available at https://drive.google.com/file/d/1RmxuFvk6TCCFxSuUuUss9cUVKI3zHOwV/view?usp=sharing.

[4] 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.

[5] 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.

[6] L. Li, A. Bartel, T. F. Bissyandé, J. Klein, Y. Le Traon, S. Arzt, S. Rasthofer, E. Bodden, D. Octeau, and P. McDaniel, "Iccta: Detecting inter-component privacy leaks in android apps," in *Proceedings of the 37th International Conference on Software Engineering-Volume 1*, pp. 280–291, IEEE Press, 2015.

[7] W. Klieber, L. Flynn, A. Bhosale, L. Jia, and L. Bauer, "Android taint flow analysis for app sets," in *Proceedings of the 3rd ACM SIGPLAN International Workshop on the State of the Art in Java Program Analysis*, pp. 1–6, ACM, 2014.

[8] A. Rountev and D. Yan, "Static reference analysis for gui objects in android software," in *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '14, (New York, NY, USA), pp. 143:143–143:153, ACM, 2014.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] A. S. Christensen, A. Møller, and M. I. Schwartzbach, "Precise analysis of string expressions," in *International Static Analysis Symposium*, pp. 1–18, Springer, 2003.

[16] 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.

[17] 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.

[18] "Droidbench benchmark suite.," 2020. Available at https://github.com/secure-software-engineering/DroidBench.

[19] "Icc-bench benchmark suite.," 2020. Available at https://github.com/fgwei/ICC-Bench.

[20] 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.

[21] "An insecure example application.," 2020. Available at https://github.com/hdiv/insecure-bank.