ConDySTA: Context-Aware Dynamic Supplement to Static Taint Analysis

Xueling Zhang, Xiaoyin Wang, Rocky Slavin, Jianwei Niu
Department of Computer Science, University of Texas at San Antonio, Texas, USA
{xueling.zhang, xiaoyin.wang, rocky.slavin, jianwei.niu}@utsa.edu

***Abstract*—Static taint analyses are widely-applied techniques to detect taint flows in software systems. Although they are theoretically conservative and designed to detect all possible taint flows, static taint analyses almost always exhibit false negatives due to a variety of implementation limitations. Dynamic programming language features, inaccessible code, and the usage of multiple programming languages in a software project are some of the major causes. To alleviate this problem, we developed a novel approach, DySTA, which uses *dynamic* taint analysis results as additional sources for static taint analysis. However, naïvely adding sources causes static analysis to lose context sensitivity and thus produce false positives. Thus, we developed a hybrid context matching algorithm and corresponding tool, ConDySTA, to preserve context sensitivity in DySTA. We applied REPRODROID [1], a comprehensive benchmarking framework for Android analysis tools, to evaluate ConDySTA. The results show that across 28 apps (1) ConDySTA was able to detect 12 out of 28 taint flows which were not detected by any of the six state-of-the-art static taint analyses considered in REPRODROID, and (2) ConDySTA reported no false positives, whereas nine were reported by DySTA alone. We further applied ConDySTA and FLOWDROID to 100 top Android apps from Google Play, and ConDySTA was able to detect 39 additional taint flows (besides 281 taint flows found by FLOWDROID) while preserving the context sensitivity of FLOWDROID.**

***Index Terms*—Taint Analysis, Dynamic Supplement, Context Sensitivity**

I. INTRODUCTION

Taint analysis [2], [3] can detect taint flows in software programs and has a wide range of applications in software and system security such as vulnerability detection [4]–[6], privacy leak detection [7]–[10], and malware detection [11], [12] among others. The intensive research efforts in the area generally fall into two categories: **Dynamic taint analyses** [2] propagate taints at run time through memory locations so they always find true taint flows. However, they may miss taint flows which are not triggered during testing and will cause run-time overhead if applied during production. Alternatively, **Static taint analyses** [3], propagate taints based on an overestimation of all possible program paths leading to the detection of all possible taint flows with no false negatives but some false positives due to infeasible paths.

Despite the theoretical soundness of static taint analyses, various practical complexities often lead to false negatives in real-world scenarios. As an example, our evaluation shows that while FLOWDROID, the state-of-the-art static taint analysis tool for Android apps, finds 281 taint flows in 100 top Android apps but misses at least 19 taint flows which are confirmed by dynamic taint analysis. Earlier studies [13], [14] also show the existence of false negatives in static taint analyses. A later study [15] performed an evaluation of six state-of-the-art static taint analysis tools for Android and also reported many common false negatives not detected by *any* of the evaluated tools. Such false negatives may result in undetected vulnerabilities, privacy leaks, malicious apps, etc.

The reason behind these false negatives can often be attributed to dynamic programming language features such as reflection calls in Java, dynamically loaded or generated code, external code execution through database servers and network servers, and multi-language code (e.g., native code and shell scripts). We refer to such features as *blockers* as they block the static taint analyses from tracing taint flows. Existing static taint analysis tools are either sound only with assumptions on the absence of blockers (e.g., most Java static analyses assume the absence of reflection calls / dynamically generated code [16] and do not consider taint flows through databases and files [17]), or rely on manual method summaries (e.g., FLOWDROID relies on method summaries to handle Android system calls and native calls) which are often incomplete and quickly become obsolete as code evolves.

In this paper, we propose an approach that uses the results of dynamic taint analysis as additional sources to supplement static taint analysis as a means to reduce false negatives. We implement and evaluate our approach for the Android platform because it has well-established static taint analysis tools [12], [18], [19] and downstream applications [6], [7], [9]. Although the effectiveness of such dynamic supplement is limited by the test coverage, our evaluation shows that it can reduce many false positives with a simple random testing strategy based on Monkey [20].

The base version of our approach is referred as DySTA (Dynamic Supplement of static Taint Analysis). DySTA first runs static taint analysis and dynamic taint analysis with the same set of initial sources, respectively. Once DySTA observes a variable holding a tainted value in the dynamic taint analysis that is *not* observed as tainted by

the static taint analysis, the variable will be considered a new source (referred to as an *intermediate source* to be differentiated from the original sources). For the set of all intermediate sources, DySTA runs the static taint analysis again to find additional taint flows. Unlike with static analysis, dynamic analysis is performed at run time, so it is less affected by blockers and is able to trace taint flows through dynamically loaded or generated code. Furthermore, even for pure black boxes (e.g., external flows through network servers or un-instrumented code), it is still possible to apply value-based dynamic taint analyses [21] which detect taint flows based on the observation of unique values preset at the source locations. As a result, DySTA retains the static taint analysis ability to trace all possible program paths outside of blockers which may not be triggered during testing while gaining the ability to detect traces *through* blockers thanks to the taint flows detected by dynamic taint analysis.

While the above approach can reduce false negatives, the basic design of DySTA has an important limitation. Since it simply concatenates static and dynamic taint flows without any constraints, the context sensitivity of the original static taint analysis will be lost. Therefore DySTA alone will lead to additional false positives besides those in the original static taint analysis for cases where blockers were analyzed. To overcome this, we further propose hybrid context matching in which the context of dynamic taint flows is injected into the intermediate sources. DySTA is then augmented so the subsequent static taint analysis considers only taint flows matching the injected context. By incorporating context matching, we implemented ConDySTA (Context-aware DySTA) as an extension of FlowDroid, a state-of-the-art static taint analysis tool for Android apps. We evaluated DySTA and ConDySTA with ReproDroid, a benchmarking framework for Android analysis tools [1]. The results show that both DySTA and ConDySTA were able to reduce 12 out of the 28 common false negatives missed by all six static taint analyses considered in ReproDroid, and context preservation enabled ConDySTA to further eliminate all nine additional false positives reported by DySTA. We also performed a comparison of our approach and FlowDroid on the 100 most downloaded Android apps according to PlayDrone [22]. Our evaluation showed that, with minimal testing and dynamic analysis, ConDySTA was able to detect 39 additional taint flows on top of the 281 taint flows reported by FlowDroid. Furthermore, ConDySTA was able to preserve context sensitivity and rule out 1,029 taint flows with context mismatches from the detection results of DySTA.

This paper presents the following contributions.

- We demonstrate that dynamic taint analysis results can be used as supplement to static taint analysis to reduce false negatives in practice.
- We developed a novel approach, ConDySTA, to preserve the context sensitivity of static taint analysis when supplemented by dynamic taint analysis.
- We performed evaluations using the ReproDroid benchmark and 100 top Android apps from Google Play demonstrating that ConDySTA can reduce many false negatives reported by state-of-the-art taint analysis tools and largely reduce false positives from our baseline solution.

The rest of this paper is organized as follows. We first introduce a running example and describe our motivation and high-level solution in Section II. Then we formalize our problem and introduce more details of our approach in Section III. We then describe our implementation in Section IV and our comparison with FlowDroid in Section V. Finally, we discuss the related research efforts in Section VII, before we conclude in Section VIII.

II. Running Example and Approach Overview

In this section, we present a running example to motivate and illustrate our approach.

A. Running Example

Consider the example code in Listing 1. In the code, method `foo()` simply returns the value it receives as the argument. In particular, we assume that the parameter value of `foo()` is passed to `blocker(...)`, and the value is fetched in method `foo2()` by invoking `blocker2()`, and `foo2()` returns the fetched value, which is further returned by `foo()`. Here, we do not make assumptions about the implementation of `blocker(...)` and `blocker2()`, but one example of such an implementation can be the writing and reading files in the file system or tables in a database, respectively. Such a taint flow could not be traced as we assume blocker code portions (i.e., methods `blocker(...)` and `blocker2()`) are not accessible or analyzable by static taint analysis. Therefore, static taint analyses will not taint variable `inter` in Line 6 and will thus miss the taint flow from method invocation `source()` in Line 10 to `sink(out)` at Line 13.

```
1  public String foo(String in){
2      blocker(in);
3      return foo2();
4  }
5  public String foo2(){
6      String inter = blocker2(); //an intermediate
           source
7      return inter;
8  }
9  public void bar(boolean flag){
10     String in = source(); //an original source
11     String out = foo(in);
12     if(flag){
13         sink(out); //a potential taint flow
14         String in2 = "safe";
15         sink(foo(in2)); //a false positive
16     }
17 }
```

Listing 1: Static Taint Analysis False Negative Example

B. DySTA Approach

Our basic solution, DySTA, executes the program and performs *dynamic* taint analysis after the initial *static* taint analysis. In the example, DySTA would taint variable

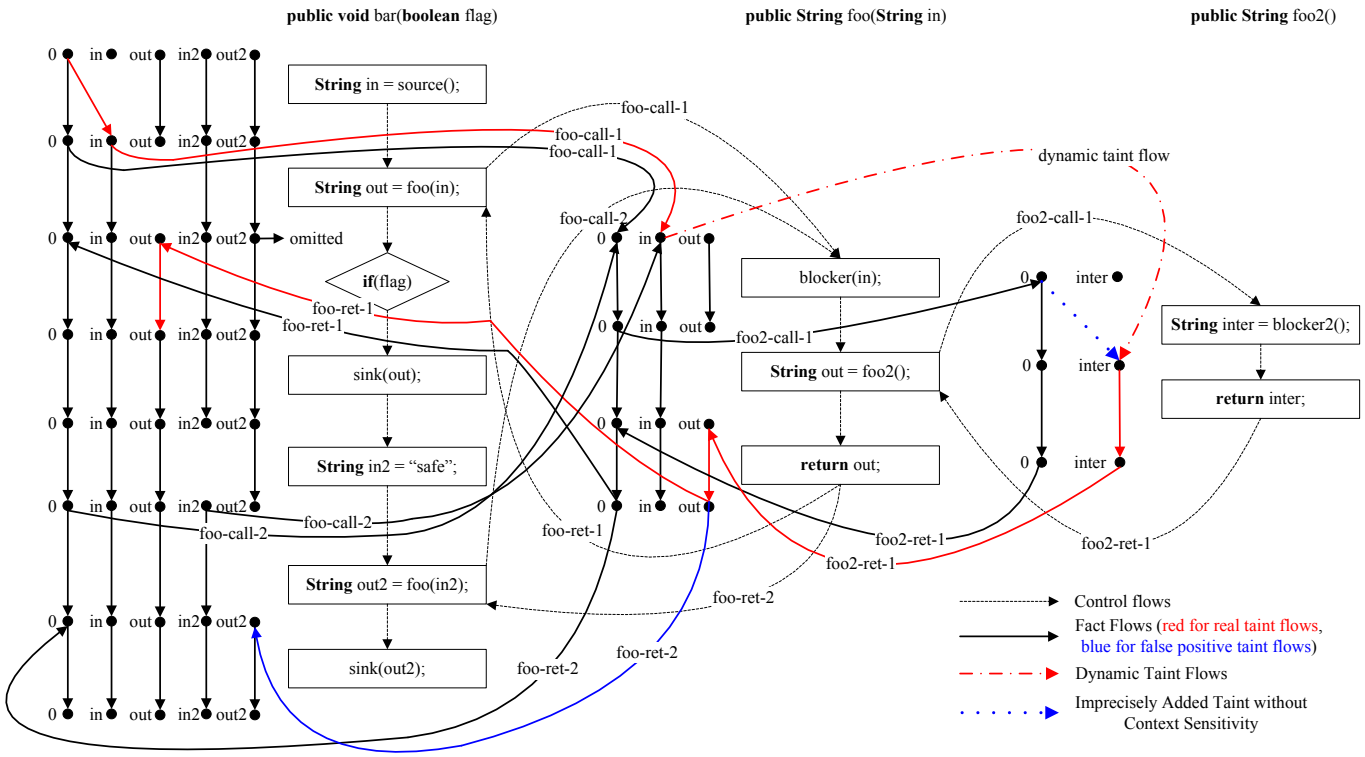

Fig. 1: Analysis of the Running Example using IFDS Framework

`inter` at Line 6 as an intermediate source according to the result of the dynamic taint analysis which is able to follow the data flow through methods `blocker(...)` and `blocker2()`. Static analysis would then be applied again incorporating the intermediate source, thus detecting the taint flow at Line 13. However, since DySTA would not consider the calling context of `foo(...)` and `foo2()`, the taint would be further propagated to the expression `foo(in2)` at Line 15, although the argument `in2` passed in here is not a user information value from the original source method invocation at Line 10. Therefore, DySTA will detect an additional taint flow at Line 15. This false positive would be due to the second static taint analysis which incorporates the intermediate source from the dynamic analysis which does not include the calling context. So while the static taint analysis itself is context sensitive, the combination of dynamic taint analysis and static taint analysis becomes partially context-insensitive.

It should be noted that, because dynamic taint analysis cannot cover all possible paths, static taint analysis may be necessary to detect the taint flow at Line 13 (i.e., when parameter `flag` is not true during the execution). Furthermore, the lack of execution coverage on Lines 13-15 would make it impossible to rule out the false positive at Line 15 based on dynamic analysis alone (i.e., finding out `foo(in2)` at Line 15 is returning value "safe").

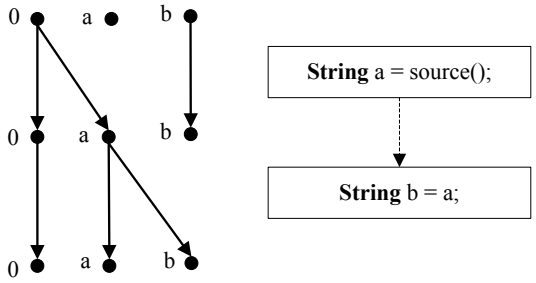

Fig. 2: Illustration of Taint Flow Functions

C. Code Analysis with the IFDS Framework

The IFDS framework, developed by Reps, Horwitz and Sagiv [23], defines a general mechanism to perform inter-procedural, flow-sensitive, and context-sensitive analysis. The framework is based on a program's inter-procedural control flow graph, referred as the "exploded super graph". The exploded super graph of our running example is presented in Figure 1. In Figure 1, we use dashed arrows to present control flows. Cross-procedure control flows are decorated with labels such as "foo-call-1", "foo-ret-1", and "foo-call-2" to differentiate call sites. For example, we can tell from the labels that call edge "foo-call-1" matches with return edge "foo-ret-1".

IFDS uses flow functions to represent transfer functions in flow-sensitive analysis on distributive finite properties. A flow function consists of a set of "from facts" and "destination facts", as well as arrows from the former to the latter. An arrow from fact a in the "from facts" to fact

b in the "destination facts" indicates that if a holds before the statement is executed, b will hold after the statement is executed.

For example, Figure 2 shows flow functions of static taint analysis in which the facts are local variables (indicating that the variable is tainted or not), plus 0, a special fact that always holds. For statement `String a = source();`, the arrow from fact 0 to fact a indicates that variable `a` will be tainted no matter what (as 0 always holds). The arrow from fact b to fact b indicates that if b is tainted before the statement, then it is still tainted after its execution. Similarly, for statement `String b = a;`, the arrow from a to a indicates that whether a is tainted is unchanged before and after the statement, and the arrow from a to b indicates that if a is tainted before the statement execution, b will be tainted afterward. Given flow functions of all statements in the exploded super graph, the inference of a fact at a certain statement can be deduced to a graph reachability problem. In particular, it is a CFL reachability problem [24] because along the reachability path the arrows labeled with call-sites and return-sites must match to preserve context sensitivity.

In Figure 1, we show the flow functions of all statements in the three methods as solid arrows to the left of the control flow graph. Note that for method `bar(boolean)`, we omitted the fact for variable `flag` and the flow functions (and control flow) of the else branch to enhance the readability of the graph. From the figure, we marked as red the edges that form the taint flow from method invocation `source()` to the method invocation `sink(out)`. This flow cannot be detected by IFDS because it contains a dynamic taint flow path (presented as the red dash-dotted arrow on the top left) through `blocker(String)` and `blocker2()`, which cannot be statically analyzed at all. Without dynamic taint flow, IFDS finds no flows from the source to the sinks.

It should also be noted that if we simply add the dynamic taint flow path as an additional flow as shown in the graph, IFDS will still not identify the taint flow (marked in red), because the return edge "foo2-ret-1" will be mismatched with "foo-call-1" in this flow, and this flow is actually not along a feasible execution path as it directly goes from `foo(String)` to `foo2()`. Another possible solution is to add the whole dynamic execution paths inside `blocker(String)`, `blocker2()`, and their dependencies into the exploded super graph. However, since the code inside blockers are out of the box of the original static analysis, their transfer functions (i.e., flow functions) may be undefined. This can make the implementation of combined analysis very complicated and even infeasible. From figure 1, we can also see that, if we directly use `inter` as the source (i.e., adding the dotted blue arrow from fact 0 to fact *inter*), IFDS will identify the two flows to both `sink(out)` (true positive, marked in red) and `sink(out2)` (false positive, marked in blue), because IFDS allows unmatched call/return sites (feasible paths) but disallows mismatched call/return sites (infeasible paths).

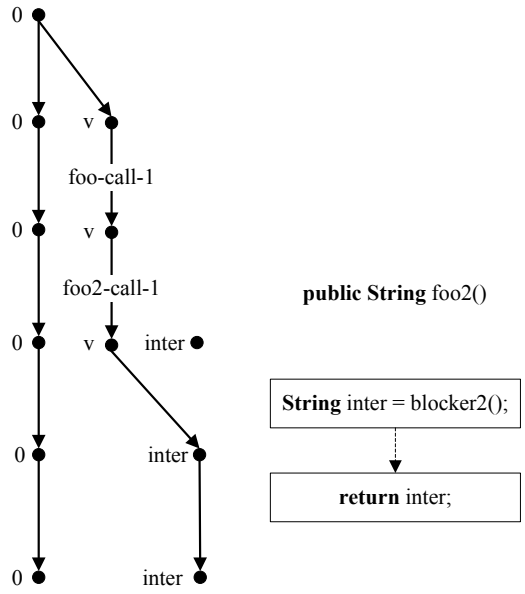

Fig. 3: Illustration of ConDySTA Solution

D. Incorporating Context

In ConDySTA, we inject the dynamic calling context of an intermediate source s to the static taint analysis from s. In particular, the dynamic calling context of an intermediate source s consists of all the call-sites that have not returned on the dynamic taint propagation path from the original source to s. In the calling context, the call-sites are ordered in the same order as they are in the dynamic taint path. In our running example, the dynamic calling contexts of intermediate source `inter` at Line 6 will be `foo(in) at Line 11`, and `foo2()` at Line 3.

With the acquired dynamic calling context of s, in the following static taint analysis from s, ConDySTA will filter out the static taint propagation paths that do not match with the dynamic calling context. This is not a straightforward process due to recursive calls (for which there can be infinite static taint propagation paths). In particular, in the CFL-reachability [24] algorithm to solve the IFDS problem [23], besides finding feasible paths with matched call-site-return-site pairs (so that the paths are feasible with context sensitivity), we need to further identify the feasible paths containing a sequence of unmatched return-sites that match with the dynamic calling context C. We refer to such static taint-propagation paths as C-context-matching paths.

ConDySTA implements this by extending the exploded super graph in IFDS framework with a virtual flow to the intermediate source with the dynamic calling context as the edges. In the extended graph, we can directly apply the standard CFL-reachability algorithm, and each taint propagation path from the original source in the extended graph can be mapped to a C-context-matching path. Figure 3 shows the solution of ConDySTA on the

running example. From the figure we can see that we added a virtual fact v, that taints the intermediate source *inter* and a number of virtual arrows on v before it taints *inter*. These virtual arrows are labeled with call-edges in the dynamic calling context, and are added in the reverse order so that they can match the return edges during the following IFDS analysis.

E. Lack of Dynamic Taint-Propagation Paths

Typical dynamic taint analyses propagate taints along with read/write accesses to memory locations along with program execution, so it is natural for them to record the dynamic taint-propagation paths. However, even this recording can sometimes be difficult in practice.

First of all, if the taint propagation is at the OS/hardware level [2], [25], it can be difficult to map the taint propagation paths back to source code due to multiple levels of abstractions. Even if a mapping is constructed, the mapping can be fragile and specific to a version of programming language runtime and OS. Second, the dynamic taint analysis itself may still miss some taint paths through file systems, databases, and networks. Third and most importantly, unlike static taint analyses which are based on relatively stable programming language syntax/semantics, dynamic taint analyses need to work with most fine-grained system features and implementation details, so they can be easily out-of-date due to fast software evolution. For example, there have been two major dynamic taint analysis frameworks for the Android system: TaintDroid [2] and TaintART [26]. Neither of them support analysis on Android system versions above Android 6 (currently 8 and 9 are the most common Android versions [27]). Therefore, the simpler value-based dynamic taint analyses [21] often has better applicability. In particular, value-based dynamic taint analyses detect taint flows by inserting taints into the value fetched at the original sources (e.g., replacing the fetched value with strange values indicating the source location), or by changing data values at source locations in different executions and monitoring correlated value changes at other locations.

In all of these cases, ConDySTA may face a situation where the dynamic taint analysis can provide only tainted code locations, but not the taint propagation paths from the sources. So for the code in Listing 1, we can tell that variable `inter` at Line 7 is tainted, but we may not tell where the taint comes from and cannot extract the dynamic calling context from the dynamic taint-propagation path. To handle such cases, ConDySTA takes advantage of a key observation that *the dynamic calling context* of an intermediate source s is always a sub-sequence of the call stack trace of s. So we can directly extract the dynamic calling context from the stack trace, which is almost always accessible in dynamic taint analyses. For our example, the call stack trace for the intermediate source is as below.

```
at method foo2() (Line 7)
at method foo(String) (Line 3)
at method bar(boolean) (Line 11)
at some method (some line)
...
```

In the stack trace, the first three items actually provide the dynamic calling context: a call-site of `foo2()` at Line 3 of method `foo(String)` and a call-site of `foo(String)` at Line 11 of method `bar(boolean)`. We can see that not all items of the stack trace belong to the dynamic calling context. For example, the call-site of `bar(boolean)` is not part of the dynamic calling context, because `source()` is invoked inside/after it, so the call edge for `bar(boolean)` does not need to be matched. On the other hand, if a call-site belongs to dynamic calling context, all call-sites above it in the stack trace are part of the dynamic calling context as the source value must go through these call-sites to reach the intermediate source as arguments, global variables, or value containers in blockers.

The key challenge is to decide how long a prefix of the stack trace needs to be in the dynamic calling context. The basic idea is that, if a call-site belongs to the dynamic calling context, it must be executed after the source location. Therefore, we can determine whether a call-site belongs to the dynamic calling context by checking the call stack of the source location or checking for tainted values in the reachable memory from the call-site.

III. Approach

In this section, we will first introduce the algorithm for DySTA and then present the construction algorithm for dynamic calling contexts in ConDySTA for propagation-based dynamic taint analysis. Finally, we will describe how dynamic calling contexts can be extracted for value-based dynamic taint analysis.

Before describing the approach, we provide the following static and dynamic taint analysis definitions. In our definitions, we use the term *expression location* to describe a pair of the form (*expr*, *line*) where *expr* is an expression and *line* is a description of where the expression is read or written in the code. For example, an expression location in our running example is (`inter`, Line 6).

Definition 1: **Static Taint Analysis** We define a static taint analysis as a function STA: ($Code$, $Srcs$) $\rightarrow$ $TaintLocs$, where $Code$ is the code base to be analyzed and $Srcs$ are the set of expression location in $Code$ serving as the sources. $TaintLocs$ are a set of expression locations in $Code$.

Definition 2: **Propagation-Based Dynamic Taint Analysis** We define a propagation-based dynamic taint analysis as a function D_p: ($Code$, $Inputs$, $Srcs$) $\rightarrow$ $Paths$, where $Code$ and $Srcs$ are as defined in Definition 1, and $Inputs$ are input used to execute the code base. $Paths$ are a set of *taint propagating program paths*. Each path p in $Path$ is in the form of (s_1, s_2, ..., s_n), where $\exists$ $src \in Srcs$ such that s_1 reads src, and $\exists$ $i \in Input$ such that p is a contiguous sub-sequence of $exec(Code, i)$ (representing

Algorithm 1 DySTA Algorithm

Input:
Code is the code base to analyze
Srcs is the set of source locations
Inputs is the set of inputs for dynamic analysis
Output:
TaintLocs is a set of tainted locations

```
1: TaintLocs ← STA(Code, Srcs)
2: Paths ← D_p(Code, Srcs, Inputs)
3: interSrcs ← ∅
4: for all p ∈ Paths do
5:   for all s_i ∈ p do
6:     if ¬blocked(s_i) ∧ blocked(s_{i−1}) then
7:       for all expression locations t ∈ s_i do
8:         if tainted(t) ∧ t ∉ TaintLocs then
9:           Add t to interSrcs
10:        end if
11:      end for
12:    end if
13:  end for
14: end for
15: NewTaintLocs ← STA(Code, interSrcs)
16: TaintLocs ← TaintLocs ∪ NewTaintLocs
```

the execution path of *Code* with input i), and the taint can be transitively propagated on p.

Definition 3: **Value-Based Dynamic Taint Analysis** We define a value-based dynamic taint analysis as a function D_v: (*Code*, *Inputs*, *Srcs*) $\rightarrow$ *LocStacks*, where *Code*, *Srcs*, and *Inputs* are as defined in Definition 2. *LocsStacks* are a set of pairs in the form of (*loc*, *stack*), where *loc* is an expression location that holds tainted value at least once in the execution, and *stack* is a corresponding call stack when *loc* holds a tainted value.

It should be noted that for both propagation-based and value-based dynamic taint analysis, one expression location may be tainted multiple times, and ConDySTA considers them as different intermediate sources if they have different taint propagating program paths or call stacks, because they may have different dynamic calling contexts which lead to different context matching in the following static taint analysis.

A. DySTA Algorithm

Based on the definitions above, our algorithm for DySTA is presented in Algorithm 1. The basic idea behind the algorithm is to first identify intermediate sources from the results of dynamic taint analysis (Lines 1-14), and then apply static taint analysis using them as sources (Lines 15-16). In particular, we first fetch the results of static taint analysis using original sources (Line 1), fetch the results of dynamic taint analysis (Line 2), and initialize the set of intermediate sources (Line 3). Then, for each statement in each taint-propagating execution path p (Lines 4-5), we first check whether the statement is re-entering statically analyzable code (Line 6). If so, DySTA checks which

Algorithm 2 Construction Dynamic Calling Context

Input:
path is a taint propagating program path
InterSrcs is the set of intermediate sources
Output:
ContextMap is a Hashmap from intermediate sources on *path* to their corresponding dynamic calling context

```
1: DContext ← ∅
2: ContextMap ← ∅
3: for all s_i ∈ p do
4:   for all expression locations t ∈ s_i do
5:     if t ∈ InterSrcs then
6:       ContextMap.Put(t, DContext.copy())
7:     end if
8:   end for
9:   if isCallSite(s_i) then
10:    DContext.push(s_i)
11:  else if isReturnSite(s_i) then
12:    DContext.pop()
13:  end if
14: end for
```

expression locations in that statement are tainted (Lines 7-8), and add those tainted expression locations to the set of intermediate sources (Line 9).

DySTA extracts intermediate sources from only the statements re-entering statically analyzable code (referred to as *re-enter statements*) to avoid useless intermediate sources. In a statically analyzable segment of p, a taint on an earlier statement can be also statically propagated to tainted expression locations in later statements. Therefore, if static taint analysis using tainted expression locations in an earlier statement generates $result_e$, and static taint analysis using tainted expression locations in a later statement generates $result_l$, $result_e$ will be a strict super set of $result_l$. Thus, there is no need to extract intermediate sources from later statements. For similar reason, in Line 8, we do not consider as intermediate sources the expression locations that are already tainted by the original static taint analysis STA. In other words, we consider only the dynamic taint flows through blockers, which are not detectable by static taint analyses.

B. Dynamic Calling Context and Graph Extension

For ConDySTA, we extend DySTA with the matching of dynamic calling contexts. In particular, at Line 15 of DySTA algorithm, before calling STA to perform IFDS-based static taint analysis, ConDySTA inserts two processes. The first process extracts dynamic calling contexts for each intermediate source, and the second process extends the exploded super graph to add the dynamic calling context to it (see Section II-D and Figure 3). We present the algorithm we use to construct dynamic calling context from taint propagation paths as Algorithm 2.

The algorithm walks along the taint-propagating execution path (Lines 3-4), and collect all call-sites that have

not returned in a stack *DContext* (Lines 9-13). When an intermediate source t is reached (Line 5), ConDySTA copies the current *DContext* and save it as t's dynamic calling context.

C. ConDySTA for Value-based Taint Analyses

When the taint-propagating execution path is not available (e.g., in value-based taint analysis), we cannot take advantage of the path to fetch the intermediate sources and the dynamic calling context. In such a case, we directly use the expression locations detected to hold tainted values as intermediate sources, and their call stack trace as dynamic calling contexts, as explained in Section II-E. The challenge is to determine how many levels in the call stack trace (denoted as $stack_i$) belong to the dynamic calling context. Since only the open call-sites executed after the original source location need to be matched along the taint path, only items executed after the original source location need to be identified in the call stack trace.

If the source location is known, we can instrument the source location and fetch its call stack trace $stack_s$. Then we compare $stack_s$ and $stack_i$ to extract their common post-fix *post*. We can see that call-sites in *post* are not-yet-returned call sites executed before the original source location, so $stack_i \setminus post$ will be the dynamic calling context to be matched. If the source location is not known, we cannot use the solution above. In this case, we can instrument all call sites in $stack_i$, and scan the reachable memory locations at the call site to check whether the tainted value can be observed. If the tainted value exists, we consider the call-site to be a part of the dynamic calling context, as the call-site should be executed after the source location is executed.

IV. Implementation

In our implementation of ConDySTA, we use FlowDroid [3] for static taint analysis, as it is a state-of-the-art tool based on IFDS framework, and is compatible with the most updated Android system and apps. For dynamic taint analysis, we use value-based dynamic taint analysis, because the state-of-the-art propagation-based tools [2], [26], [28] are all out-of-date and do not work with Android 6.0 or higher (Android now is at 10). Although having the weakness of not handling control dependencies and encrypted data, value-based dynamic taint analysis also has its advantage on handling pure black boxes (e.g., web APIs whose implementations are on remote servers). Note that ConDySTA can always take advantage of new dynamic taint analysis once they are available. Figure 4 shows the implementation of DySTA and ConDySTA. They both first collect intermediate sources with dynamic analysis, and then detect additional taint flows using static taint analysis from intermediate sources. ConDySTA additionally checks whether an additional taint flow has a calling context matching with the dynamic calling context of the corresponding intermediate source.

A. User Profile For Tainted Values

Value-based dynamic taint analysis requires tainted values for sources. Specifically, we use the values in the user profile of an Android device as the tainted values. The information type and taint values are presented in Table IV in the Appendix.

B. Intermediate Source Collection

When collecting the intermediate sources, we instrument all return values of methods whose return types are `java.lang.String`. The reason is that all the tainted values are of string type and are stored in string variables. Although they are sometimes organized as fields in objects, there is often a method declared in the object's class to fetch the value of the sensitive data as a string. Due to performance concern, we only implemented the return value. In further research, we may apply static analysis or machine learning to select part of string-type parameters as instrumentation points.

After instrumentation, we rebuild the smali code back into APK format for testing. We use the Android Debug Bridge (adb) to automatically install the rebuilt apps onto our test device, login with predefined profile if required, and use Monkey [20] to explore the app for 20 seconds. We use minimal testing in the implementation and evaluation of ConDySTA to check whether it can detect additional taint flows even with minimal testing. So our evaluation results actually show a lower estimation of the ability of ConDySTA, and equipping ConDySTA with more advanced testing may further enhance its effectiveness. During testing, we utilize the Android system log to record the return values and call stacks of String type methods. Table I shows an example where Line 1 shows the return value; Line 2 shows the method that be invoked (`com.facebook.internal.AttributionIdentifiers.getAndroidAdvertiserId`). The following lines show the call stack trace of this method. We consider a method as an intermediate source when its return value contains any user info in Table IV. In this example, the return value is the AdvertiserId, so we consider `getAndroidAdvertiserId()` as an intermediate source. We also check for concatenated, reversed and hashed format of the user info. For example, "355458061189396_ZX1G22KHQK" is a concatenation form of IMEI and Serial number.

Due to the essential weakness of value-based dynamic taint analysis, we will miss encrypted values. Please note that this can be resolved if ConDySTA is integrated with a propagation-based dynamic taint analysis tool (which is straightforward once such a tool is available). Furthermore, the taint flow of encrypted values are usually of less concern.

C. Applying FlowDroid

We run FlowDroid with the original sources to detect statically tainted locations and rule them out intermediate sources, this reduces the source locations for the second

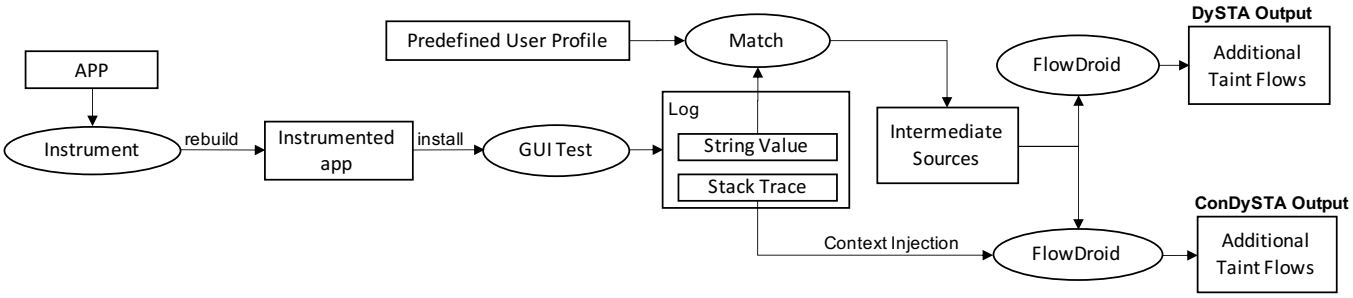

Fig. 4: Implementation of ConDySTA

1	09-12 16:25:13.442 W System.err: java.lang.Exception: fc1303d8-7fbb-44d8-8a68-a79ffac06fea
2	09-12 16:25:13.443 W System.err: at com.facebook.internal.AttributionIdentifiers.getAndroidAdvertiserId (AttributionIdentifiers.java:1)
3	09-12 16:25:13.443 W System.err: at com.facebook.marketing.internal.RemoteConfigManager.run (RemoteConfigManager.java:5)
4	09-12 16:25:13.443 W System.err: at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1133)
5	09-12 16:25:13.443 W System.err: at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:607)
6	09-12 16:25:13.443 W System.err: at java.lang.Thread.run(Thread.java:761)

TABLE I: System log of String method

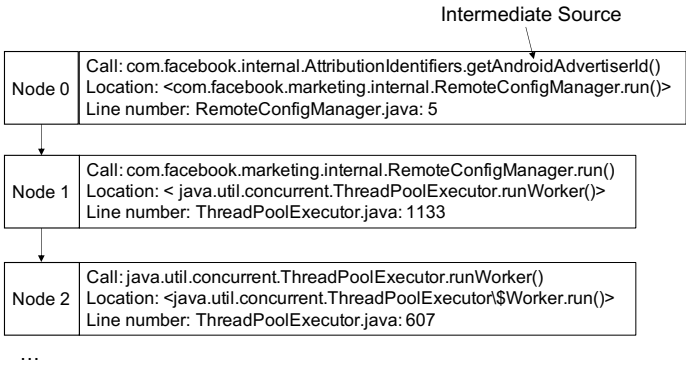

Fig. 5: ConDySTA Call Path

round of static analysis and makes sure that ConDySTA always finds new taint flows. After we collected the intermediate sources, we feed these sources to FlowDroid as new source locations, and run FlowDroid again.

Context Match. Using the generated full path, we can perform context matching with the call stack trace. First, we convert the call stack trace to a call path. For the example call stack trace in Table I, we generate the call path as Figure 5. Then, we will convert this call path to the form of exploded super graph as shown in Figure 3, and combine it with the exploded super graph generated by FlowDroid before running it for static taint analysis.

V. Evaluation

In our evaluation, we consider two data sets. The first data set is a part of ReproDroid [15], a large and up-to-date benchmark which combines multiple earlier benchmarks [3], [10], [29] for static taint analysis tools for Android apps. ReproDroid's data set primarily consists of small apps with labeled taint flows (i.e., ground truth) written by researchers. The second data set consists of real-world apps from the Google Play store. We selected the 100 most downloaded apps that could be instrumented and successfully analyzed by FlowDroid according to PlayDrone [22], a collection of meta data for Android apps on the Google Play store. The full list of the apps (as well as ConDySTA's implementation and detailed results) are available at our anonymized project website[1].

A. Research Questions and Summarized Answers

- **RQ1:** How many more taint flows can ConDySTA detect than static taint analysis alone? 28 taint flows across 28 apps from the benchmark were not detected by *any* of the six state-of-the-art static taint analysis tools. Of those 28 common false negatives, ConDySTA was able to detect 12. Among the 100 real-world apps, ConDySTA detected 39 more taint flows than FlowDroid across 12 apps. FlowDroid detected a total of 281 taint flows across 57 apps in total. The tainted information included email addresses, country, language, device's manufacturer, advertising ID, user's full name and username.
- **RQ2:** How many false positives can context sensitivity preservation reduce compared to naïve dynamic supplementation? In the benchmark evaluation, DySTA detected 21 taint flows (12 true positives and nine false positives) and ConDySTA reduced *all* of the nine false positive without missing any true positives. In the real-world app evaluation, ConDySTA was able to remove 1,029 taint flows with mismatched context from the result of DySTA.
- **RQ3:** Does ConDySTA detect taint flows not detected by the dynamic taint analysis itself? Among the 39 taint flows detected by ConDySTA, 19 of them were also directly detected (and thus confirmed) by dynamic taint analysis, the remaining 20 were additionally detected through static taint analysis.

[1]https://sites.google.com/view/condysta2020

- **RQ4:** How efficient is ConDySTA? The execution time of ConDySTA ranges from less than one second (when no intermediate sources are found) to 4,266 seconds, which is comparable to the execution time of FlowDroid.

B. Evaluation on the Benchmark

A major challenge in evaluating program analysis tools on real-world applications is the lack of ground truth, so we first evaluate ConDySTA on the ReproDroid [1] benchmark, which consists of apps with taint flows labeled by earlier researchers. ReproDroid combines three existing benchmarks: DroidBench [30], ICCBench [29], and DIALDroidBench [10] and contains additional apps with code features not covered by the three benchmarks. The apps in ReproDroid are mostly written by earlier researchers and are simple enough for the researchers to manually identify and label all taint flows (i.e., pairs of source and sink locations[2]) in the app. The apps cover many different code features to check whether static analysis tools can handle handle those features. In the initial study [1] on ReproDroid, the authors evaluated six state-of-the-art static taint analysis tools: Amandroid [12], DIALDroid [19], DidFail [31], DroidSafe [32], FlowDroid [18], IccTA [33], and reported detailed results.

1) Reducing False Negatives: In order to evaluate the effectiveness of ConDySTA on detecting additional taint flows, we applied ConDySTA on the apps (from ReproDroid) which contain at least one taint flow that cannot be detected by any of the six static taint analysis tools (i.e., a *common false negative* of all six tools). We consider these apps and taint flows because users can always combine existing static taint analysis tools and use the union of their results to reduce false negatives, and we want to check whether ConDySTA can further reduce false negatives on top of that. We identified 33 common false negative taint flows from 33 apps (one common false negative per app). Their covered code features include implicit taint flows, native code, reflection, and inter component communication. Among the 33 apps, five of them were out-of-date and thus could not be installed or crashed immediately upon execution (note that none of the labeled taint flows in these apps are observed), so they were excluded. We applied ConDySTA on the remaining 28 apps, and it detected 12 correct taint flows and thus reduced 12 out of the 28 common false negatives. We also applied DySTA on the 28 apps and it detected the exact same taint flows, thus showing that by adding context sensitivity, ConDySTA did not introduce false negatives.

Table II shows the details of these 12 taint flows. In the table, the four columns represent their IDs in the benchmark, covered code features, enclosing apk names, and source/sink pairs. In flow 124 (ImplicitFlow), the data has been converted into an array of `Char` and copied back to another string before flowing to the sink. In flows 191 and 192 (Native code), native code is used to fulfil part of the taint flow. In flows 203 and 206-209, part of the taint flow (sending information through intents) is fulfilled with method invocations performed by reflection along with dynamic generation of class/method signatures used in reflection. In flows 24, 25, 27 and 32, the data is transmitted across components through intents. Notably, the latter two code features (Reflection and ICC) are supported by some of the tested tools (e.g., Amandroid [12], DIALDroid [19], and IccTA [33]), but they are only partially supported so some complicated cases cannot be handled as shown in the result of ReproDroid study [1]. ConDySTA detected these 12 flows based on intermediates sources such as `de.ecspride.ImplicitFlow1: java.lang.String copyIMEI(java.lang.String)`, `android.content.Intent: java.lang.String getStringExtra(java.lang.String)`, etc. From the result, we can also see that ConDySTA is independent from code features (i.e., types of blockers), so it reduces common false negatives caused by various types of blockers.

ConDySTA failed to reduce the remaining 16 common false negatives, simply because the corresponding taint flows do not involve any string type return values (which are the only instrumentation points of ConDySTA) and thus ConDySTA fails to detect intermediate sources. If ConDySTA instrumented all string type parameters, it would be able to detect all 28 of the common false negatives. However, we did not implement ConDySTA to instrument all string parameters due to the high overhead (due to the need to extract call-stacks at all instrumentation points) in real-world apps, resulting in a lack of scalability. In real-world apps, taint flows are much longer and more complicated so a string type return value is more likely to be involved. Upon further research, we may select only part of string-type parameters and other-type variables as instrumentation points, which may realize the full potential of ConDySTA.

2) False Positives: To evaluate ConDySTA's performance on reducing false positive caused by context insensitivity, we applied both DySTA and ConDySTA on 43 apps from ReproDroid that contained at least one true negative. When constructing ReproDroid and earlier benchmarks, researchers also labeled fake taint flows (i.e., pairs of sources and sinks without a flow between them). These labeled true negatives can be used to check whether static analysis tools report false positives. Of the 186 such fake taint flows in the 43 apps, DySTA mistakenly reported nine of them. With context matching, ConDySTA did not report any of them, so it reduced nine false positives of DySTA to zero. Note that ConDySTA is implemented to supplement a static taint analysis (i.e., FlowDroid), and the static taint analysis itself may report false positives which are not caused by ConDySTA.

[2]Multiple flows between the same pair of source and sink locations are considered as one.

ID	Feature	Apk	Source & Sink
DroidBenchExtend			
124	ImplicitFlows	ImplicitFlow1	android.telephony.TelephonyManager.getDeviceId()
			android.util.Log.i(java.lang.String,java.lang.String)
191	Native	SinkInNativeLibCode	android.telephony.TelephonyManager.getDeviceId()
			mod.ndk.ActMain.cFuncSendData(java.lang.String)
192	Native	SourceInNativeCode	mod.ndk.ActMain.cFuncGetIMEI(android.content.Context)
			android.telephony.SmsManager.sendTextMessage(java.lang.String, ...)
203	Reflection_ICC	OnlyIntent	android.telephony.TelephonyManager.getDeviceId()
			android.telephony.SmsManager.sendTextMessage(java.lang.String, ...)
206	Reflection_ICC	OnlyTelephony	java.lang.reflect.Method.invoke(java.lang.Object,java.lang.Object[])
			android.telephony.SmsManager.sendTextMessage(java.lang.String, ...)
207	Reflection_ICC	OnlyTelephony_Dynamic	java.lang.reflect.Method.invoke(java.lang.Object,java.lang.Object[])
			android.telephony.SmsManager.sendTextMessage(java.lang.String, ...)
208	Reflection_ICC	OnlyTelephony_Reverse	java.lang.reflect.Method.invoke(java.lang.Object,java.lang.Object[])
			android.telephony.SmsManager. sendTextMessage(java.lang.String, ...)
209	Reflection_ICC	OnlyTelephony_Substring	java.lang.reflect.Method.invoke(java.lang.Object,java.lang.Object[])
			android.telephony.SmsManager.sendTextMessage(java.lang.String, ...)
ICCBench			
24	IccTargetFinding	icc_dynregister1	android.telephony.TelephonyManager.getDeviceId()
			android.util.Log.d(java.lang.String,java.lang.String)
25	IccTargetFinding	icc_dynregister2	android.telephony.TelephonyManager.getDeviceId()
			android.util.Log.d(java.lang.String,java.lang.String)
27	IccTargetFinding	icc_explicit1	android.telephony.TelephonyManager.getDeviceId()
			android.util.Log.d(java.lang.String,java.lang.String)
32	IccTargetFinding	icc_implicit_mix1	android.telephony.TelephonyManager.getDeviceId()
			android.util.Log.d(java.lang.String,java.lang.String)

TABLE II: False negative taint flows detected by ConDySTA

App Package Name	Size (KLOC)	FlowDroid	DySTA	ConDySTA (Dynamic)	ExecTime(s) DySTA+ ConDySTA	ExecTime(s) FlowDroid
com.amazon.mShop.android.shopping	10881	1	25	2(0)	25	257
com.dianxinos.dxbs	3034	1	77	15(6)	4266	1162
com.disney.WMWLite	1489	2	11	2(2)	357	131
com.forthblue.pool	1778	3	22	2(0)	1630	270
com.gameloft.android.ANMP. GloftDMHM	2540	20	3	3(0)	29	18
com.mxtech.videoplayer.ad	4044	3	4	1(1)	574	27
com.pinterest	5534	0	2	4(4)	95	138
com.sgiggle.production	6015	0	1	1(0)	44	32
com.tubitv	7660	0	5	3(2)	38	273
com.waze	2996	1	1	1(0)	16	115
org.mozilla.firefox	2155	24	74	4(4)	18	1265
paint.by.number.pixel.art.coloring. drawing.puzzle	4795	0	14	1(0)	23	64
...	...	...	...	...	...	
Total	N/A	281	1068	39(19)	N/A	

TABLE III: Taint flows detected by ConDySTA in real-world apps

C. Evaluation on Real World Apps

Five of the six tools used in the ReproDroid benchmark could not be applied to our real-world app dataset as four of them (Amandroid, DidFail, DroidSafe, and IccTA) do not execute on recent apps[3] [1], and one (DIALDroid) targets only inter-app taint flows and not general intra-app taint flows. For these reasons, we were able to benchmark ConDySTA against those apps using ReproDroid's compatible test set, but we could *not* use ReproDroid to compare ConDySTA's performance against these tools for modern apps. To address this and present a more complete evaluation, we also evaluated ConDySTA on current real-world apps and compared it with the remaining working tool, FlowDroid, to test ConDySTA's relevance to the current app landscape.

For fair comparison, we count taint flows the same way as FlowDroid. In particular, multiple taint flows between the same pair of source and sink locations are counted as one taint flow. So even if ConDySTA detects a different taint flow for a pair of source and sink locations between which FlowDroid already detects a flow, we do not consider ConDySTA to have found a new taint flow. Furthermore, we use the configuration of FlowDroid with context sensitivity and least false negatives (FlowDroid has some configurations sacrificing soundness for performance). Finally, we make sure ConDySTA and FlowDroid use the same set of sources and sinks. Column 2 of Table IV shows the sources we use for each user information type. Note that full name, user name and password are provide through user input,

[3]They support up to Android API level 19, and Android API is currently at level 29

so we use the `EditText.getText()` method invocations of the corresponding UI widget as the sources. To further confirm, we instrumented the `EditText.getText()` method invocations and print out the value passed in to make sure our input values are caught by these sources.

We present our evaluation results on additionally-detected taint flows in Table III. In the table, Columns 1-5 present the name of the app, the size of the app in thousands of lines of smali code (note that we have only the byte code of apps as they are closed source), the number of taint flows detected by FlowDroid, the additional number of taint flows detected by DySTA, the additional number of taint flows detected by ConDySTA (with the number of taint flows that also detected by dynamic taint analysis within these flows in brackets), and the execution time. Note that as we have 100 apps and limited space, we present only the apps with at least one taint flow detected by ConDySTA. The full results are available in Tables V and VI in the Appendix. Also, for the execution time, we include only the following context-aware static taint analysis portion. Since the execution time of dynamic taint analysis largely depends on the testing intensity (and we are using minimal testing in our evaluation), it does not make much sense to combine the execution time.

1) Additionally Detected Flows Over FlowDroid: Among the 100 apps tested, FlowDroid detected 281 taint flows using the Android platform sources, while ConDySTA detected 39 more taint flows. 19 of these 39 were confirmed with dynamic taint analysis and eight of the remaining can be manually confirmed (see Section V-C5 for more detailed inspection results). From Table III, we can see that these 39 flows are distributed over 12 different apps. This shows that the practical complexity that causes unsoundness of static taint analysis is very common among top Android apps. For some of the apps (e.g., com.dianxinos.dxbs), FlowDroid detects zero or very few taint flows while ConDySTA detected many, which shows that ConDySTA may be very helpful for some apps where blockers are used intensively.

2) ConDySTA vs. DySTA: A comparison between Columns 4 and 5 in Table III shows the benefit of ConDySTA. In particular, ConDySTA reduced 1,029 context-mismatched taint flows from 49 apps. So we can see that the reduction of context-mismatched taint flows happens in almost all of the apps. It should be noted that a context mismatched taint flow may not necessarily be fake. In very rare cases, the taint flow may happen under a different context not covered by dynamic taint analysis or even through another intermediate source not observed in dynamic taint analysis. However, we believe they should be removed because they should not be inferred from observed facts of the dynamic taint analysis. As an analogy, a weather forecaster may have a flaw so that Wednesday's weather is always reported to be stormy, which could be true in rare cases, but the flaw and corresponding forecasting results should be removed because they are not results from the forecasting model (which may be imperfect by itself). Note that in our evaluation on ReproDroid, all the removed context-mismatched taint flows are fake flows.

3) Comparison with Pure Dynamic Taint Analysis: We further studied whether ConDySTA detects only the taint flows that are already detected by dynamic taint analysis. If so, its value would be diminished. Among the 39 taint flows detected by ConDySTA, we instrumented the sink methods and applied dynamic taint analysis to check how many taint flows could be detected. The results are presented in the brackets of Column 5 in Table III, which shows that 19 taint flows can be detected (and thus confirmed as true positives) and the remaining 20 cannot be detected. This shows that ConDySTA does provide more value by performing static taint analysis from the intermediate sources.

4) Execution Time: Finally, we recorded the execution time of ConDySTA (see Column 6 of Table III). We can see that the execution time is within 5,000s, and for most of the apps it ranges from several hundred seconds to thousands of seconds. This is similar to those of FlowDroid. Notably, as ConDySTA invokes FlowDroid for intermediate sources, the largest portion of its execution can be attributed to FlowDroid. It should be noted that DySTA+ConDySTA sometimes take much longer time than simply running FlowDroid because of the additional intermediate sources.

5) Qualitative Analysis: To understand why FlowDroid has the false negatives that ConDySTA detected, we further performed a qualitative analysis on the taint flows detected by ConDySTA but not FlowDroid. Among the 39 taint flows, 23 flows are in apps which are heavily obfuscated and we were not able to understand the full taint paths (Note that 11 of the 23 flows were confirmed in dynamic taint analysis). Among the remaining 16 flows that we managed to fully understand, six flows were missed by FlowDroid because the data flowed through the network (sent to remote servers and fetched back), four flows were not detected because the data flowed to local cache files and were later read back, and six flows were not detected due to FlowDroid's flawed modeling of `HashMap.putAll()`, which we confirmed with a trivial app with only this function on the taint path. Note that `HashMap` is particularly difficult to handle in static analysis as it can easily create many false positives if the entire `HashMap` is conservatively tainted. Finally, we can see that the blockers in real-world apps are very different from those pre-defined in ReproDroid. So ConDySTA's independence of blocker types can be an important benefit when applied to real-world apps.

D. Threats to Validity

One major threat to the internal validity comes from value-based taint analysis. Due to coincident string matches, some of the detected false negatives may not

be real false negatives. To reduce such threat, we use complicated profile data to avoid coincident matches, and manually confirmed all detected false negatives on ReproDroid, and a large portion of those from real-world Android apps. One major threat to the external validity comes from the size and variety of our subject apps. To reduce such threat, we consider both a large existing benchmark and top real-world Android apps.

VI. Discussion

Generality on Dynamic Taint Analysis. Since ConDySTA needs only intermediate sources (nodes on the taint paths) and their calling contexts (method invocations along the taint paths) from the dynamic taint analysis, ConDySTA should be able to directly take the output of any propagation-based dynamic taint analysis. Even if the method invocations along the taint paths are not provided by the dynamic taint analysis tool (which is unlikely for propagation-based analysis), ConDySTA can still directly use the system stack traces at intermediate sources as estimated calling contexts (just as how it handles value-based dynamic taint analysis). So, once a new dynamic taint analysis framework becomes available, ConDySTA can easily take advantage of it.

Generality on Static Taint Analysis. ConDySTA uses FlowDroid as the static analysis tool to be supplemented because it is context-sensitive, very robust to be still able to handle most Android apps on the market, and has been adopted by many downstream research efforts (e.g., [7] and [9]). DySTA integrates with static taint analysis by providing intermediate sources as new sources, so it can be directly used with almost any static taint analysis tools (as long as they allow adding new sources) without any effort. ConDySTA further encodes calling context into the inter-procedure control-flow graph in the IFDS framework, so it can be directly integrated to any IFDS-based static taint analysis. ConDySTA can be further adapted to integrate with more broader categories of static taint analyses by encoding the calling context into the intermediate code representation the analyses are based on.

VII. Related Works

We discuss related works in three categories: taint analysis, coping with unsoundness of static analysis, and static supplement of dynamic analysis.

A. Taint Analyses for Android

Our approach supplement static taint analysis with dynamic taint analysis results, so it is related to existing static and dynamic taint analysis techniques. Here we limit our discussion for Android due to the large number of existing work in the area. FlowDroid [3] is a state-of-art static information analysis tool for Android apps. Other Android-oriented static information analysis techniques include CHEX [34], LeakMiner [35], and ScanDroid [36]. Specifically, CHEX [34] detects component hijacking vulnerabilities in Android applications by tracking taints between externally accessible interfaces and sensitive sources or sinks. LeakMiner [35] is an earlier context-insensitive information-flow analysis technique for detecting privacy leaks in Android apps. ScanDroid [36] tracks taint flows among multiple apps and detects privacy leaks into other apps. There are also dynamic taint analysis techniques such as TaintDroid [2] and CopperDroid [37] that perform OS-level or application-level of dynamic taint propagation. TaintArt [26] and TaintMan [28] further extends the existing dynamic taint analysis to support Android RunTime (ART) which adopts ahead-of-time compilation strategy and replaces previous virtual-machine-based Dalvik. Jung et al., [21] proposed PrivacyOracle, which uses differential analysis of tainted values perform dynamic taint analysis on black-box systems without instrumenting the application or the underlying OS. Tripp et al. [38] utilized Bayesian reasoning to determine if an information release at a sink point represents a privacy leak. It calculates the possibility of legitimate information releases at a sink based on the distance between the information about to be released and the original sensitive data. Continella et al. [39] proposed a black-box analysis tool to detect privacy leaks in mobile apps by analyzing network traffic. All of these taint analyses are either static or dynamic and they all suffered from the limitations of static analysis or dynamic analysis. ConDySTA provides a general approach to use dynamic taint analysis results in static taint analysis so it can take advantage of all these existing taint analyses or newly developed taint analyses in the future.

B. Tackling Practical Unsoundness of Static Analysis

Prior researchers have already noticed the unsoundness of static analysis in practice. Researchers from Coverity [40] explained the challenges of applying static analysis to real world [41], and they mentioned in the paper that the static inaccessibility to code as one of the major challenges. In academia, different dynamic supplements of static analysis have been proposed. On handling reflections, Livshits et al. [42] proposed an approach to statically infer information about reflective call sites from program code. TamiFlex [16] perform dynamic analysis to record destinations of reflection calls and use such records to supplement the program call graph, which is the basis for many static analyses. DroidRA [43], in contrast, uses static constant propagation to estimate potential reflection call destinations in Android apps. On handling dynamically loaded / generated code, Wei and Ryder [44] developed blended taint analysis for JavaScript which summarizes dynamically generated code from dynamic analysis output and perform static taint analysis based on the summaries. Averroes [45] generates mock libraries with analysis summaries so it can be used for replacement of missing libraries. Dufour [46] proposed to collect calling structure data at run time, and feed it as input to static method-

escape analysis, so that some complicated code portions can be analyzed more efficiently. PRuby [47] by Furr et al. is a static-type inference system for the Ruby programming language. It uses dynamic profiles to handle the three dynamic language features in Ruby: send, require and eval, which performs reflection invocations, dynamic code loading, and dynamic code generation, respectively. ConDySTA is different from all the above works because it is not specific to any types of blockers and can combine off-the-shelf static and dynamic taint analysis, while these works cope with pre-defined blockers (mainly dynamic code features). Our evaluation show that blockers in real-world apps can be very different from the code-feature blockers considered in ReproDroid, so ConDySTA's independence of blocker types is an important advantage over existing works when applied in practice.

C. Static Supplement to Dynamic Analyses

The third category of research efforts use static analysis results to guide or supplement dynamic analysis to support certain code features or to enhance efficiency. To protect users from cross-site scripting (XSS) attacks, Vogt et al. [48] proposed a dynamic taint analysis framework to monitor sensitive information flows within the web browser. The framework has a complementary static analysis to be invoked when necessary to detect indirect control flow dependencies which are not handled by taint propagation in the dynamic analysis. Concolic analyses [49] also fall into this category. Several later efforts [50]–[52] start from a seed dynamic execution, and try to generate legal or more test cases by statically analyzing dependencies among elements on the executed trace. Christakis et al. [53] further proposed an approach which takes advantage of static analysis to identify some verified paths, and guide the dynamic execution to only unverified paths. Zheng et al. [54] first used static analysis to collect the activity path towards the sensitive API and then use dynamic analysis to trigger the UI activity path. Another large category of works use static analysis to determine where to add run-time checks and thus reduce the number of checks and run-time overhead. For example, Rhodes et al. [55] used static analysis to reduce runtime checks for data racing by coalescing checks and compressing shadow locations. Sengupta et al. [56] proposed EnfoRSer, which first statically partitions code into a number of statically bounded regions, and checks whether these statically bounded regions are executed atomically at run time. There are also efforts on static-analysis-guided dynamic analysis [50] that try to generate tests dynamically confirming a statically detected defect, such as DSDCrasher [57] and Check'n'crash [58]. HARVEST [59] is a hybrid approach to extract runtime values in the case of obfuscation and anti-analysis techniques. For a predefined interesting data, they statically collect the program slices which contain invocation of the interesting points. For each slice, conditional statements were removed to avoid anti-analysis feature like time and logic bombs. In the dynamic analysis stage, HARVEST execute the slice code and report target interesting values. They evaluated HARVEST on malware and identified reflective method invocation on sensitive methods, which is not detectable by static or dynamic analysis tool. Wong et al. [60] proposed to detect and reverse language-based obfuscation via dynamic instrumentation. Ahmand et al. [61] proposed to automatically targeted triggering the method of interest (MOI), which used Inter Component Communications (ICC) for passing data between components. Then they extracted runtime values of reflection and encrypted strings. Xia et al. [62] proposed to reduce the false positives of static analysis approaches by verifying the detected leaks through an approximated dynamic analysis. Compared with these efforts, ConDySTA works on the opposite direction that uses dynamic analysis results to alleviate practical unsoundness of static analysis, and also injects dynamic context into the static taint analysis.

VIII. Conclusion

In this paper, we explored the use of dynamic taint analysis as a supplement to static taint analysis to reduce false negatives. We demonstrated the potential loss of context sensitivity in such an approach and developed a hybrid context matching mechanism to retain it. We further implemented ConDySTA for value-based dynamic taint analysis as an augmentation for FlowDroid, and evaluated it on the ReproDroid benchmark and 100 top Android apps. Our evaluation showed that ConDySTA was able to reduce 12 of 28 common false negatives present in all existing tools from ReproDroid and detect 39 additional taint flows from 100 apps which were not detectable by FlowDroid. Furthermore, the use of context-sensitivity preservation helped remove all nine false positives and 1,029 context-mismatching taint flows reported by the baseline solution DySTA.

We believe that the general idea of using dynamic analysis to reduce false positives in static analysis is promising, so we plan to work in the following directions. First, we plan to evaluate ConDySTA on a larger set of apps and applying ConDySTA to the combinations of other dynamic taint analyses and static taint analyses on or beyond Android. Second, we plan to work on the dynamic supplementation of other static analyses (e.g., type-state analysis, points-to analysis) with practical unsoundness while preserving analysis properties. Third, we plan to explore techniques for guiding dynamic executions through blockers to better reduce false negatives in static analysis.

Acknowledgment

This work is supported in part by NSF Awards NSF-1846467, NSF-1736209, NSF-2007718, and NSF-1948244.

References

[1] F. Pauck, E. Bodden, and H. Wehrheim, "Do android taint analysis tools keep their promises?" in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2018, pp. 331–341.

[2] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth, "Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones," in *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'10, 2010, pp. 1–6.

[3] 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," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2014, pp. 259–269.

[4] N. Jovanovic, C. Kruegel, and E. Kirda, "Pixy: A static analysis tool for detecting web application vulnerabilities," in *2006 IEEE Symposium on Security and Privacy (S&P'06)*. IEEE, 2006, pp. 6–pp.

[5] G. Wassermann and Z. Su, "Sound and precise analysis of web applications for injection vulnerabilities," in *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2007, pp. 32–41.

[6] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang, "Chex: statically vetting android apps for component hijacking vulnerabilities," in *Proceedings of the 2012 ACM conference on Computer and communications security*, 2012, pp. 229–240.

[7] J. Huang, Z. Li, X. Xiao, Z. Wu, K. Lu, X. Zhang, and G. Jiang, "{SUPOR}: Precise and scalable sensitive user input detection for android apps," in *24th {USENIX} Security Symposium ({USENIX} Security 15)*, 2015, pp. 977–992.

[8] Y. Nan, M. Yang, Z. Yang, S. Zhou, G. Gu, and X. Wang, "Uipicker: User-input privacy identification in mobile applications," in *24th {USENIX} Security Symposium ({USENIX} Security 15)*, 2015, pp. 993–1008.

[9] X. Wang, X. Qin, M. B. Hosseini, R. Slavin, T. D. Breaux, and J. Niu, "Guileak: Tracing privacy policy claims on user input data for android applications," in *Proceedings of the 40th International Conference on Software Engineering*. ACM, 2018, pp. 37–47.

[10] A. Bosu, F. Liu, D. Yao, and G. Wang, "Collusive data leak and more: Large-scale threat analysis of inter-app communications," in *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, 2017, pp. 71–85.

[11] Y. Feng, S. Anand, I. Dillig, and A. Aiken, "Apposcopy: Semantics-based detection of android malware through static analysis," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2014, pp. 576–587.

[12] (2017) Amandroid. [Online]. Available: https://bintray.com/arguslab/maven/argus-saf/3.1.2

[13] B. Livshits, M. Sridharan, Y. Smaragdakis, O. Lhoták, J. N. Amaral, B.-Y. E. Chang, S. Z. Guyer, U. P. Khedker, A. Møller, and D. Vardoulakis, "In defense of soundiness: a manifesto," *Communications of the ACM*, vol. 58, no. 2, pp. 44–46, 2015.

[14] D. King, B. Hicks, M. Hicks, and T. Jaeger, "Implicit flows: Can't live with 'em, can't live without 'em," in *International Conference on Information Systems Security*. Springer, 2008, pp. 56–70.

[15] F. Pauck, E. Bodden, and H. Wehrheim, "Do android taint analysis tools keep their promises?" in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2018, pp. 331–341.

[16] E. Bodden, A. Sewe, J. Sinschek, H. Oueslati, and M. Mezini, "Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders," in *Proceedings of the 33rd International Conference on Software Engineering*. ACM, 2011, pp. 241–250.

[17] A. Dasgupta, V. Narasayya, and M. Syamala, "A static analysis framework for database applications," in *2009 IEEE 25th International Conference on Data Engineering*. IEEE, 2009, pp. 1403–1414.

[18] 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," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2014, pp. 259–269.

[19] (2017) Daildroid. [Online]. Available: https://github.com/dialdroid-android/DIALDroid

[20] A. Developers, "Ui/application exerciser monkey," 2012. [Online]. Available: https://developer.android.com/studio/test/monkey.html

[21] J. Jung, A. Sheth, B. Greenstein, D. Wetherall, G. Maganis, and T. Kohno, "Privacy oracle: a system for finding application leaks with black box differential testing," in *Proceedings of the 15th ACM conference on Computer and communications security*. ACM, 2008, pp. 279–288.

[22] (2018) Playdron metadata. [Online]. Available: https://archive.org/details/android_apps&tab=about

[23] T. Reps, S. Horwitz, and M. Sagiv, "Precise interprocedural dataflow analysis via graph reachability," in *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, 1995, pp. 49–61.

[24] T. Reps, "Program analysis via graph reachability," *Information and software technology*, vol. 40, no. 11-12, pp. 701–726, 1998.

[25] E. B. Nightingale, D. Peek, P. M. Chen, and J. Flinn, "Parallelizing security checks on commodity hardware," in *ACM Sigplan Notices*, vol. 43, no. 3. ACM, 2008, pp. 308–318.

[26] M. Sun, T. Wei, and J. Lui, "Taintart: A practical multi-level information-flow tracking system for android runtime," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2016, pp. 331–342.

[27] A. Developers, "Top android os versions," 2020. [Online]. Available: https://www.appbrain.com/stats/top-android-sdk-versions

[28] W. You, B. Liang, W. Shi, P. Wang, and X. Zhang, "Taintman: an art-compatible dynamic taint analysis framework on unmodified and non-rooted android devices," *IEEE Transactions on Dependable and Secure Computing*, 2017.

[29] F. Wei, S. Roy, and X. Ou, "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*, 2014, pp. 1329–1341.

[30] 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," *Acm Sigplan Notices*, vol. 49, no. 6, pp. 259–269, 2014.

[31] (2015) Didfail. [Online]. Available: https://www.cert.org/secure-coding/tools/didfail.cfm

[32] (2015) Droidsafe. [Online]. Available: https://mit-pac.github.io/droidsafe-src/

[33] (2016) Iccta. [Online]. Available: https://sites.google.com/site/icctawebpage/source-and-usage

[34] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang, "Chex: Statically vetting android apps for component hijacking vulnerabilities," in *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, 2012, pp. 229–240.

[35] Z. Yang and M. Yang, "Leakminer: Detect information leakage on android with static taint analysis," in *Proceedings of the 2012 Third World Congress on Software Engineering*, 2012, pp. 101–104.

[36] A. P. Fuchs, A. Chaudhuri, and J. S. Foster, "Scandroid: Automated security certification of android applications," *Manuscript, Univ. of Maryland, http://www.cs.umd.edu/avik/projects/scandroidascaa*, vol. 2, no. 3, 2009.

[37] K. Tam, S. J. Khan, A. Fattori, and L. Cavallaro, "Copperdroid: Automatic reconstruction of android malware behaviors," in *22nd Annual Network and Distributed System Security Symposium*, 2015.

[38] O. Tripp and J. Rubin, "A bayesian approach to privacy enforcement in smartphones," in *23rd {USENIX} Security Symposium ({USENIX} Security 14)*, 2014, pp. 175–190.
[39] A. Continella, Y. Fratantonio, M. Lindorfer, A. Puccetti, A. Zand, C. Kruegel, and G. Vigna, "Obfuscation-resilient privacy leak detection for mobile apps through differential analysis." 2017.
[40] "Coverity," https://scan.coverity.com/.
[41] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, and D. Engler, "A few billion lines of code later: using static analysis to find bugs in the real world," *Communications of the ACM*, vol. 53, no. 2, pp. 66–75, 2010.
[42] B. Livshits, J. Whaley, and M. S. Lam, "Reflection analysis for java," in *Asian Symposium on Programming Languages and Systems*. Springer, 2005, pp. 139–160.
[43] L. Li, T. F. Bissyandé, D. Octeau, and J. Klein, "Droidra: Taming reflection to support whole-program analysis of android apps," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*. ACM, 2016, pp. 318–329.
[44] S. Wei and B. G. Ryder, "Practical blended taint analysis for javascript," in *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, 2013, pp. 336–346.
[45] K. Ali and O. Lhoták, "Averroes: Whole-program analysis without the whole program," in *European Conference on Object-Oriented Programming*. Springer, 2013, pp. 378–400.
[46] B. Dufour, B. G. Ryder, and G. Sevitsky, "Blended analysis for performance understanding of framework-based applications," in *Proceedings of the 2007 international symposium on Software testing and analysis*, 2007, pp. 118–128.
[47] M. Furr, J.-h. D. An, and J. S. Foster, "Profile-guided static typing for dynamic scripting languages," in *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, ser. OOPSLA '09. New York, NY, USA: ACM, 2009, pp. 283–300. [Online]. Available: http://doi.acm.org/10.1145/1640089.1640110
[48] P. Vogt, F. Nentwich, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna, "Cross site scripting prevention with dynamic data tainting and static analysis."
[49] K. Sen, D. Marinov, and G. Agha, "Cute: a concolic unit testing engine for c," in *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 5. ACM, 2005, pp. 263–272.
[50] S. Zhang, D. Saff, Y. Bu, and M. D. Ernst, "Combined static and dynamic automated test generation," in *Proceedings of the 2011 International Symposium on Software Testing and Analysis*. ACM, 2011, pp. 353–363.
[51] K. J. Hoffman, P. Eugster, and S. Jagannathan, "Semantics-aware trace analysis," *ACM Sigplan Notices*, vol. 44, no. 6, pp. 453–464, 2009.
[52] D. Babić, L. Martignoni, S. McCamant, and D. Song, "Statically-directed dynamic automated test generation," in *Proceedings of the 2011 International Symposium on Software Testing and Analysis*. ACM, 2011, pp. 12–22.
[53] M. Christakis, P. Müller, and V. Wüstholz, "Guiding dynamic symbolic execution toward unverified program executions," in *Proceedings of the 38th International Conference on Software Engineering*. ACM, 2016, pp. 144–155.
[54] C. Zheng, S. Zhu, S. Dai, G. Gu, X. Gong, X. Han, and W. Zou, "Smartdroid: an automatic system for revealing ui-based trigger conditions in android applications," in *Proceedings of the second ACM workshop on Security and privacy in smartphones and mobile devices*, 2012, pp. 93–104.
[55] D. Rhodes, C. Flanagan, and S. N. Freund, "Bigfoot: static check placement for dynamic race detection," *ACM SIGPLAN Notices*, vol. 52, no. 6, pp. 141–156, 2017.
[56] A. Sengupta, S. Biswas, M. Zhang, M. D. Bond, and M. Kulkarni, "Hybrid static–dynamic analysis for statically bounded region serializability," in *ACM SIGPLAN Notices*, vol. 50, no. 4. ACM, 2015, pp. 561–575.
[57] C. Csallner, Y. Smaragdakis, and T. Xie, "Dsd-crasher: A hybrid analysis tool for bug finding," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 17, no. 2, p. 8, 2008.
[58] C. Csallner and Y. Smaragdakis, "Check'n'crash: combining static checking and testing," in *Proceedings of the 27th international conference on Software engineering*. ACM, 2005, pp. 422–431.
[59] S. Rasthofer, S. Arzt, M. Miltenberger, and E. Bodden, "Harvesting runtime values in android applications that feature anti-analysis techniques." in *NDSS*, 2016.
[60] M. Y. Wong and D. Lie, "Tackling runtime-based obfuscation in android with {TIRO}," in *27th {USENIX} Security Symposium ({USENIX} Security 18)*, 2018, pp. 1247–1262.
[61] M. Ahmad, V. Costamagna, B. Crispo, and F. Bergadano, "Teicc: targeted execution of inter-component communications in android," in *Proceedings of the symposium on applied computing*, 2017, pp. 1747–1752.
[62] M. Xia, L. Gong, Y. Lyu, Z. Qi, and X. Liu, "Effective real-time android application auditing," in *2015 IEEE Symposium on Security and Privacy*. IEEE, 2015, pp. 899–914.

Appendix

In our appendix, we present the full profile data (used in value-based taint analysis) in Table IV. We also present the full results of ConDySTA's output and execution time compared with FlowDroid on 100 real-world Android apps in Tables V and VI. In the table, for better readability, we leave a cell empty if its corresponding number of leaks is zero.

User Info	Source for FlowDroid
IMEI = "355458061189396"	android.telephony.TelephonyManager: java.lang.String getDeviceId()
Serial = "ZX1G22KHQK"	android.os.Build: java.lang.String getSerial () android.telephony.TelephonyManager: java.lang.String getSimSerialNumber()
AndroidID = "a54eccb914c21863"	android.provider.Settings.Secure: java.lang.String getString(android.content.ContentResolver,java.lang.String) android.provider.Settings.System: java.lang.String getString(android.content.ContentResolver,java.lang.String)
Email = "********@gmail.com"	android.accounts.AccountManager: android.accounts.Account[] getAccounts() android.accounts.AccountManager: android.accounts.Account[] getAccountsByType(java.lang.String)
PassWord = "******" UserName = "******"	android.os.UserManager: java.lang.String getUserName() android.widget.TextView: java.lang.CharSequence getText() android.widget.EditText: android.text.Editable getText() android.widget.TextView: android.text.Editable getEditableText()
language = "English"	java.util.Locale: java.lang.String getDisplayLanguage() java.util.Locale: java.lang.String getDisplayLanguage(java.util.Locale) java.util.Locale: java.lang.String getLanguage() java.util.Locale: java.util.Locale getDefault()
country = "US"	<java.util.Locale: java.lang.String getCountry() java.util.Locale: java.lang.String getDisplayCountry(java.util.Locale) java.util.Locale: java.lang.String getDisplayCountry() android.location.Address: java.lang.String getCountryName() java.util.Locale: java.util.Locale getDefault()
AdvertiserId = "fc1303d8-7fbb-44d8-8a68-a79ffac06fea"	com.google.android.gms.ads.identifier.AdvertisingIdClient.Info: java.lang.String getId ()
timezone_1 = "CST" timezone_2 = "Central Standard Time"	com.android.exchange.utility... getTimeZoneDateFromSystemTime(byte[],int) com.android.calendar.Utils: java.lang.String getTimeZone(android.content...) com.android.calendar.CalendarUtils$TimeZoneUtils:... getTimeZone(...) java.util.Calendar: java.util.TimeZone getTimeZone() java.util.TimeZone: java.util.TimeZone getTimeZone(java.lang.String) java.util.TimeZone: java.util.TimeZone getDefault() com.adobe.xmp.impl.XMPDateTimeImpl: java.util.TimeZone getTimeZone() android.util.TimeUtils: java.util.TimeZone getTimeZone(int,boolean,long...) java.text.DateFormat: java.util.TimeZone getTimeZone()
Manufacturer = "motorola"	android.os.Build.MANUFACTURER
NetWork = "Wi-Fi"	android.net.NetworkInfo: java.lang.String getTypeName()

TABLE IV: User Info and Corresponding Source

App Package Name	Size (KLOC)	FlowDroid	DySTA	ConDySTA (Dynamic)	ExecTime(s) DySTA+ ConDySTA	ExecTime(s) FlowDroid
art.coloringpages.paint.number.zodiac.free	4348		11		1402	16
com.abtnprojects.ambatana	6094	2	4		620	75
com.adobe.reader	2084	3	45		4841	114
com.amazon.mShop.android.shopping	10881	1	25	2(0)	25	257
com.appsci.sleep	4815	7	5		65	807
com.arlo.app	7178	1	33		1946	197
com.audible.application	7531				2319	2383
com.audiomack	6796	1			18	19
com.aviary.android.feather	2579	5	31		1761	134
com.bbm	8208	1			20	2318
com.bfs.papertoss	2089	7	7		125	32
com.bydeluxe.d3.android.program.starz	5022		6		3319	10
com.calm.android	6352	1	1		209	487
com.cbs.app	8355				19	16
com.chewy.android	2873				23	20
com.classdojo.android	6088	1			19	120
com.cleanmaster.mguard	8771	15	11		2993	536
com.clearchannel.iheartradio.controller	8188				24	25
com.contextlogic.wish	2943	2	19		6082	1259
com.creativemobile.DragRacing	5630	3	4		4069	310
com.creditkarma.mobile	4594		8		14	47
com.devuni.flashlight	2371	1	11		1762	28
com.dianxinos.dxbs	3034	1	77	15(6)	4266	1162
com.discord	3238		8		922	105
com.disney.WMWLite	1489	2	11	2(2)	357	131
com.domobile.applock	2393	2	7		1268	31
com.dropbox.android	5656				56	36
com.drweb	2393	1			1868	26
com.duolingo	4309	1	15		347	191
com.ebay.mobile	8050				21	27
com.enflick.android.TextNow	9949		1		20	151
com.espn.score$_c enter$	966				26	203
com.facebook.mlite	2326	5			20	361
com.fingersoft.hillclimb	4468		11		36	21
com.forthblue.pool	1778	3	22	2(0)	1630	270
com.fox.now	5085		7		25	39
Total	N/A	281	1068	39(19)	N/A	N/A

TABLE V: Taint flows detected by ConDySTA in real-world apps

App Package Name	Size (KLOC)	FlowDroid	DySTA	ConDySTA (Dynamic)	ExecTime(s) DySTA+ ConDySTA	ExecTime(s) FlowDroid
com.game.JewelsStar	2946	2	6		45	12
com.game.SkaterBoy	2571	2	14		80	14
com.gameloft.android.ANMP.GloftDMHM	2540	20	3	3(0)	29	18
com.gameloft.android.ANMP.GloftIAHM	1596	49	40		2158	19
com.gau.go.launcherex	6996	8	41		919	140
com.gau.go.launcherex.gowidget.weatherwidget	4065	8	41		564	171
com.gonoodle.gonoodle	2736	1			17	18
com.goodrx	3883		3		76	160
com.gotv.nflgamecenter.us.lite	6308	2			17	324
com.groupme.android	2942	7	3		77	394
com.grubhub.android	4995		18		26	404
com.hulu.plus	5101				22	304
com.ibotta.android	8898				1521	14
com.imangi.templerun	2425				27	16
com.imangi.templerun2	2403		3		36	13
com.indeed.android.jobsearch	2066				33	39
com.kakao.story	3471	1	8		2150	237
com.konylabs.capitalone	5125	3			236	152
com.life360.android.safetymapd	5844	2	1		51	53
com.mcdonalds.app	8329				21	39
com.microsoft.appmanager	12102	2			26	250
com.microsoft.office.outlook	8169				74	90
com.mxtech.videoplayer.ad	4044	3	4	1(1)	574	27
com.naver.linewebtoon	6744		11		28	37
com.netflix.mediaclient	4682	3			35	7
com.offerup	8054				43	128
com.outfit7.talkinggingerfree	7787				25	14
com.outfit7.talkingtom	7990				30	20
com.outfit7.talkingtom2free	7958				36	26
com.pandora.android	13149				28	19
com.particlenews.newsbreak	3787	4	16		1016	222
com.picsart.studio	10604	1			41	41
com.pinterest	5534		2	4(4)	95	138
com.pof.android	3533	1	2		34	22
com.popshow.yolo	2801	1			16	67
com.poshmark.app	5163		4		16	13
com.postmates.android	2947		87		1203	515
com.roidapp.photogrid	6867	1	25		3340	750
com.roku.remote	4133				18	82
com.rovio.angrybirdsseasons	1814	1	7		33	23
com.sgiggle.production	6015		1	1(0)	44	32
com.shootbubble.bubbledexlue	1460	6	33		1609	187
com.skype.raider	2563	2			19	15
com.squareup.cash	3654		5		28	293
com.supercell.clashofclans	1141		3		14	3
com.supercell.hayday	1323		3		14	4
com.surpax.ledflashlight.panel	3101	1	31		1165	31
com.tencent.mm	13678	1	1		42	49
com.topfreegames.bikeracefreeworld	4423				70	21
com.tubitv	7660		5	3(2)	38	273
com.UCMobile.intl	6924	7			31	46
com.venmo	4018	3	111		9820	1064
com.viber.voip	2741	12	10		3876	1813
com.waze	2996	1	1	1(0)	16	115
com.yahoo.mobile.client.android.mail	5851				13	441
com.zillow.android.zillowmap	5331				15	5
flipboard.app	3406				569	291
jp.naver.line.android	13113	19	18		126	77
me.pou.app	1923	7			22	159
org.mozilla.firefox	2155	24	74	4(4)	18	1265
paint.by.number.pixel.art.coloring.drawing.puzzle	4795		14	1(0)	23	64
scratch.lucky.money.free.real.big.win	4571	1	30		1220	349
us.ozteam.bigfoot	4628		10		22	426
vStudio.Android.Camera360	6692	9			43	1920
Total	N/A	281	1068	39(19)	N/A	N/A

TABLE VI: Taint flows detected by ConDySTA in real-world apps, Cont.