Resolving Conditional Implicit Calls to Improve Static and Dynamic Analysis in Android Apps

JORDAN SAMHI, University of Luxembourg, Luxembourg
RENÉ JUST and MICHAEL D. ERNST, University of Washington, USA
TEGAWENDÉ F. BISSYANDÉ and JACQUES KLEIN, University of Luxembourg, Luxembourg

An implicit call is a mechanism that triggers the execution of a method m without a direct call to m in the code being analyzed. For instance, in Android apps the `Thread.start()` method implicitly executes the `Thread.run()` method. These implicit calls can be conditionally triggered by programmer-specified constraints that are evaluated at run time. For instance, the `JobScheduler.schedule()` method can be called to implicitly execute the `JobService.onStartJob()` method only if the device's battery is charging. Such conditional implicit calls can effectively disguise *logic bombs*, posing significant challenges for both static and dynamic software analyses. Conservative static analysis may produce false-positive alerts due to over-approximation, while less conservative approaches might overlook potential covert behaviors, a serious concern in security analysis. Dynamic analysis may fail to generate the specific inputs required to activate these implicit call targets. To address these challenges, we introduce Archer, a tool designed to resolve conditional implicit calls and extract the constraints triggering execution control transfer. Our evaluation reveals that ① implicit calls are prevalent in Android apps; ② Archer enhances app models' soundness beyond existing static analysis methods; and ③ Archer successfully infers constraint values, enabling dynamic analyzers to detect (i.e., thanks to better code coverage) and assess conditionally triggered implicit calls.

CCS Concepts: • **Security and privacy** → **Software security engineering**.

Additional Key Words and Phrases: Static Analysis, Android Security, Data Leaks

1 INTRODUCTION

Security and privacy in the Android ecosystem is of critical importance, because Android dominates the mobile market in number of devices [31] and number of apps [33]. Previous research has investigated static analysis [5, 17, 18, 34, 36, 51], dynamic analysis [43, 59, 68], and hybrid analysis [15, 63] of Android apps. However, malware developers evade analyzers via multiple mechanisms such as implicit calls [32] and trigger-based behavior [19, 49, 51, 54]. An implicit call triggers the execution of a method without a direct call in the code being analyzed, challenging static analyzers. For example, Android life-cycle methods like `Activity.onCreate()`, are never called explicitly in app code. A logic bomb triggers the execution of malicious code under certain circumstances, challenging dynamic analyzers. For instance, A time-related logic bomb (i.e., a time bomb) would trigger the malicious code at a given date or after a certain delay.

Static analysis works by constructing a model of code and then analyzing the model. The model must reflect all possible behaviors of the code, or else the analysis is unsound, which is unacceptable for security analysis, verification, compiler optimizations, and other contexts. On the other hand, overconservatism can result in an excess of false positive alerts. Static analysis tools typically rely on control flow graphs [1] and call graphs [46]. A dataflow analysis typically starts at an entrypoint method and propagates dataflow values along the control flow graph. When a method call is encountered, the analysis continues to other methods, using the potential target methods computed by the call graph. Call graph construction algorithms such as CHA [16], RTA [6], VTA [56], Andersen [3], Steensgaard [53], SPARK [35], etc. do not natively resolve implicit calls. Therefore, static analysis tools that rely on these algorithms are unsound

Authors' addresses: Jordan Samhi, University of Luxembourg, Luxembourg, jordan.samhi@uni.lu; René Just, rjust@cs.washington.edu; Michael D. Ernst, mernst@cs.washington.edu, University of Washington, USA; Tegawendé F. Bissyandé, tegawende.bissyande@uni.lu; Jacques Klein, jacques.klein@uni.lu, University of Luxembourg, Luxembourg.

if developers do not add ad-hoc edges: they overlook parts of the code that must be visited to ensure comprehensive analysis.

The Android framework also enables developers to explicitly *constrain* or gate the execution of implicit calls without traditional conditional statements (e.g., `if` statements). For instance, the `JobService.onStartJob()` method, which is called implicitly using method `JobScheduler.schedule()`, can be constrained to be executed only if the device battery is charging, or if the device is connected to a cellular network, etc. This is controlled by APIs of the Android framework such as `JobInfo.Builder.setRequiresCharging(true)` and `JobInfo.Builder.setRequiredNetworkType(4)`. These constraints, which malicious developers can use to trigger logic bombs, challenge dynamic analyzers, which would not execute the code targeted by these implicit calls if the constraints are not met at run time.

Previous research has made static analysis of Android apps more sound by identifying several implicit call mechanisms and proposing approaches to account for them. Implicit calls addressed in the literature include life-cycle methods [4], reflection [7, 37], callbacks [11], inter-component communication (ICC) [36], threading [4], etc. If a static analysis overlooks some implicit call mechanisms, it remains unsound and admits, for example, security vulnerabilities. Previous research also contributed to logic bomb detection in Android apps by devising static analysis and AI techniques [19, 51, 54]. However, these works are limited to detecting potential trigger-based malicious behavior and do not extract the condition under which the logic bombs are triggered. If a dynamic analysis does not execute the dormant code, the malicious code remains undetected.

We have identified a new type of implicit calls that were not previously reported in the literature: conditional implicit calls (CI calls) that can be explicitly constrained in their execution (with a conditional or a temporal trigger). Our work makes two main contributions. (1) First, we improve the soundness and precision of call graphs: (1.1) we improve soundness by adding new edges from implicit calls that were previously unknown, and (1.2) we improve precision by resolving the targets of these implicit calls. (2) Second, we extract the constraints that need to be met to execute conditional implicit calls. It is important to account for these mechanisms in Android app static analysis, since attackers can use these techniques as logic bombs [19, 51] (e.g., with time constraints) to evade static and dynamic analysis and enter the Google Play store [10, 32].

The outcome of our research can directly impact the security and privacy of apps' users. Indeed, logic bombs pose serious threats to data integrity and confidentiality, making their detection paramount for protecting users from unauthorized access and privacy breaches. By focusing on CI calls, our work improves Android apps' static models and allows for reaching these hidden threats which directly contributes to making the mobile ecosystem safer for endusers.

Our approach differs from previous approaches for handling implicit calls. FlowDroid adds a dummy main method with calls to every entrypoint, such as lifecycle methods and callbacks. The new framework calls we have identified could be treated as new entrypoints, but our approach of resolving the call targets leads to much better precision. String analysis of class/method names and of intents' content help with reflective calls and inter-component communication, but they are not effective for CI calls, which do not depend on the values of strings. Therefore, we developed a different approach to accurately model and analyze CI calls in Android apps.

The contributions of this paper are:

- A systematic search of the Android framework for implicit calls that can be triggered conditionally.
- An empirical study showing that CI calls are common.
- A novel approach to handle conditional implicit calls. It ① helps static analyzers by improving soundness and precision of the call graph, and ② extracts the constraints that need to be met in order to execute CI calls.
- An open-source implementation named Archer.

Listing 1 Code illustrating a conditional implicit call.

```
1   public class MainActivity extends Activity {
2     public static String secret;
3     @Override
4     protected void onCreate(Bundle b) {
5       super.onCreate(b);
6       secret = getSecret();
7       Constraints cs = new Constraints.Builder()
8         .setRequiredNetworkType(NetworkType.CONNECTED)
9         .setRequiresCharging(true).build();
10      OneTimeWorkRequest wr =
11        new OneTimeWorkRequest.Builder(MyWorker.class)
12        .setConstraints(cs).build();
13      WorkManager.getInstance(this).enqueue(wr);
14    }
15    private String getSecret() { /** code **/ }
16  }
17  public class MyWorker extends Worker {
18    public MyWorker(Context c, WorkerParameters w) {
19      super(c, w);
20    }
21    @Override
22    public Result doWork() {
23      String secret = MainActivity.secret;
24      leak(secret);
25      return null;
26    }
27  }
```

- A new benchmark that includes CI calls, enabling more comprehensive empirical evaluations.
- Experiments showing that Archer: ① enhances the call graph of Android apps, ② outperforms existing static analyzers, ③ extracts the conditions needed to trigger CI calls with high precision, and ④ allows dynamic analyzers to improve their code coverage.
- Our implementation, experimental scripts, and data are available at:

https://github.com/JordanSamhi/Archer
https://doi.org/10.5281/zenodo.10474310

2 MOTIVATION & BACKGROUND

This section motivates our work and presents relevant background for our work.

2.1 A Motivating Example

Listing 1 shows an example of a data leak in an Android app that uses a CI call. This example illustrates the threat model for our study. We explain why this data leak would not be detected by state-of-the-art static data leak detectors such as FlowDroid, and why dynamic analyzers might not detect it.

On line 6, a sensitive datum is stored into field `MainActivity.secret`. The call to `getSecret()` is explicit, i.e., the call site is directly connected to the `getSecret()` method implementation. Lines 7–9 build a `Constraints` object requiring that the network is connected and the device is charging. The method calls `setRequiredNetworkType`() and `setRequiresCharging`() set the *conditions* that need to be met to trigger the CI call. Then, lines 10–12 show the construction of a `setOneTimeWorkRequest` object that points to class `MyWorker` and is fed with the `constraints` object. On line 13, a `WorkManager` instance calls the `enqueue()` method with the work request as a parameter. After the call to `enqueue()`, method `MyWorker.doWork()` (lines 22–26) will be executed if the conditions set by the constraints are met. Note that **no `if` statement** is used to condition the execution of the implicit call.

If a static analysis overlooks the relationship between the `enqueue()` and `doWork()` methods and does not analyze the `doWork()` method, the analysis is unsound. This would result in missing the data leak in listing 1.

A dynamic analysis may also miss the data leak in Listing 1. This is because several conditions must be met in order to trigger the leak, such as the device being connected and charging. If these conditions are not met, the leak will not be identified. This behavior can easily be exploited by attackers to trigger logic bombs.

This concrete scenario illustrates the limitations of existing literature, i.e., these implicit mechanisms are not handled by existing static analyzers and we are the first to extract their constraints to improve dynamic analysis coverage. This paper addresses these problems by: ① augmenting call graphs with CI calls to plug a soundness hole; and ② extracting the potential conditions that must be met to execute the code for dynamic analyzers in order to improve their code coverage. We implemented these in a prototype called Archer, and section 5 evaluates the benefit of our approach.

2.2 Background and Definitions

This section introduces concepts and terminology used throughout the paper.

Explicit call of a method *m* is a method call directly referring to *m* in the code under analysis. For instance, on line 6 of listing 1, the method call `getSecret()` directly refers to method `getSecret`.

Implicit call of a method *m* triggers the execution of *m* without a direct call to *m* in the code under analysis. For instance, in listing 1, `enqueue()` implicitly calls `doWork()`, though there is no direct call to method `doWork()` in the app. Note that implicit calls exist because static analyzers are app-focused and do not, for scaling issues, analyze the Android framework, in which the entire call chain would appear and make implicit calls explicit.

Executor classes and methods trigger implicit calls. In line 13 of listing 1, class `WorkManager` is an executor class and method `enqueue()` is an executor method.

Executee. After calling an executor method, an *executee* method will be executed; its class is an *executee* class. In listing 1, `MyWorker` is an executee class. On line 22, `doWork()` is an executee method. Note that, in Android apps, some methods that we refer to as "executee" in the context of this study can be triggered without an "executor" method. For instance, a `Runnable` can be executed via a standard mechanism like `Thread thread = new Thread(myRunnable); thread.start();` which is not related to conditional implicit calls. In this study, this is out of scope as we only focus on executors that can trigger code under specific circumstances.

Helper classes and methods participate in the implicit method call mechanism, but are not executors or executees. In listing 1, the `Constraints` class on line 7 is a helper class. Examples of helper methods are `setRequiredNetworkType ()` and `setRequiresCharging ()` on lines 8–9.

Condition or constraint. In the context of this paper, a condition or constraint is a property or expression that must be satisfied in order to execute an implicit call. For instance, in Listing 1, lines 8–9 set two conditions for the execution of the implicit call that occurs at line 13: the device must be connected to a network, and the device must be charging. If the conditions are not met at run time, method `doWork()` will not be triggered.

Conditional implicit call (CI Call) is an implicit call that can be triggered conditionally without traditional conditional statements (e.g., `if` statements) in the app under analysis.

Resolve. In the context of this paper, resolving CI calls means identifying the potential targets of these calls.

3 COLLECTING METHODS ENABLING CONDITIONAL IMPLICIT CALLS

Our work provides the first in-depth list of methods enabling CI calls in the Android ecosystem, together with the methods that constrain their execution. We followed the same principled and structured methodology from previous research in this area [30, 38, 62]. This section explains it, showing how we identified the mechanisms enabling CI calls in Android. Note that, in this study, we have focused on Android API level 30, though the process described can be applied to any Android version.

3.1 Class collection

We collected classes that provide CI call mechanisms from various sources.
Community. We asked on Stack Overflow [47] for mechanisms that trigger code under given circumstances. We received one answer pointing us to the `WorkManager` class. We then performed a snowball analysis to find similar mechanisms [38]. This involved carefully reading the web page [26] associated with the `WorkManager` class to collect the classes involved in this mechanism, and following any hyperlinks that could lead us to similar mechanisms and repeating the process. This process revealed 12 classes: 2 executor classes, 2 executee classes, and 8 helper classes.
Classes analysis. We looked for job-like mechanisms. We manually examined the 493 classes whose name contains one of the following strings in the Android source code (API 30): "`trigger`", "`schedul`", "`criter`", "`execute`", "`delay`", "`work`", "`job`", "`time`". Here, "schedul" has been devised to match "schedule", "scheduler", "scheduling", etc., and "criter" has been devised to match words related to "criteria", which could hint at condition criteria.

We found 27 classes related to triggering code execution: 19 executors, 6 executees, and 2 helpers. 4 of these classes overlap with those identified through our community analysis, bringing the total number of classes of interest to 35 (i.e., $12 + 27 - 4$).
Methods analysis. We searched for ways to trigger code under time-related circumstances (temporal constraints) by manually examining the 791 methods in the Android framework that have a formal parameter whose name contains one of the following strings:

`"milli", "second", "minute", "hour", "delay", "nano", "time"`

This analysis yielded a single new executor class, increasing the number of classes of interest to 36 (i.e., $35 + 1$).
Documentation analysis. We read through the online Android guides [20] to identify mechanisms discussed in the documentation. We found 5 pages [21–25] discussing implicit call mechanisms, but all of these referred to classes and methods we had already collected.

In total, we collected 21 executor classes, 7 executee classes, and 8 helper classes.

3.2 Method collection

An analysis tool needs to work at the *method* level to improve its models and discover implicitly executed code. To do this, we carefully studied the documentation of the classes we collected and how they can be used to trigger CI calls. We found 60 executor methods, 6 executee methods, and 25 helper methods. These are unique method signatures, i.e., method families possibly consisting of multiple overriding implementations. We identified 7 execute classes but only 6 executee methods. because one method, `Runnable.run()`, is used by two executee classes: `java.util.concurrent.RunnableScheduledFuture` and `java.lang.Runnable`.

Based on the documentation, we manually produced a list of executor–executee method pairs to improve static analyzers and a list of helper methods to extract the conditions under which implicit calls might occur. Table 1 shows the types of constraints that executor classes can use to trigger executees with the help of helper methods. Most executors allow setting time-related constraints, such as specifying a delay before executing an executee or a period for executing an executee. These mechanisms, which we call temporal triggers, can serve as the basis for time bombs, which trigger

	Constraints									
	Temporal		Persistent	Sound	Device states					
					Network		Battery	Charging	Idle	Storage
Executor class	Delay	Periodic	Persistent	Sound	status	type	level	status	status	level
Timer	✓	✓								
SoundTriggerDetector				✓						
WorkManager	✓	✓			✓		✓	✓	✓	✓
JobSchedulerShellCommand	✓	✓	✓		✓	✓	✓	✓	✓	✓
JobScheduler	✓	✓	✓		✓	✓	✓	✓	✓	✓
JobSchedulerImpl	✓	✓	✓		✓	✓	✓	✓	✓	✓
JobSchedulerService	✓	✓	✓		✓	✓	✓	✓	✓	✓
JobServiceContext	✓	✓	✓		✓	✓	✓	✓	✓	✓
CompletableFuture	✓	✓								
ExecutorCompletionService										
Executor										
HandlerExecutor										
SynchronousExecutor										
RepeatableExecutor	✓	✓								
RepeatableExecutorImpl	✓	✓								
DelayableExecutor	✓	✓								
ExecutorImpl	✓	✓								
ExecutorService	✓	✓								
ScheduledExecutorService	✓	✓								
ScheduledThreadPoolExecutor	✓	✓								
AbstractExecutorService										

Table 1. Constraints that can be set on executor classes to trigger conditional implicit calls.

malicious code at a specific time. We consider these temporal constraints as a specific subset of the general constraints we found. One executor, the `SoundTriggerDetector`, can trigger code based on the sound detected by the device. Six executors allow to set constraints depending on device states: ① the status of the network, i.e., connected or not; ② the type of the network, cellular or Wi-Fi; ③ the battery level, i.e., low or not; ④ the charging status, i.e., in charge or not; ⑤ the idle status, i.e., currently idle or not; and ⑥ the storage level, i.e., low or not.

Creating these pairs — that is, resolving the executors — improves precision of the call graph, compared to treating each executee as an unrestricted entrypoint. As new API methods are added to Android, our methodology can be rerun or extended to expand Archer's current mapping.

Our mapping of executors-to-executees makes static analysis of Android apps more sound by introducing links in the app's call graph that were previously nonexistent.

4 APPROACH

Archer is a tool that improves the accuracy of static and dynamic Android analyzers by accounting for conditional implicit calls and extracting the conditions necessary for their execution, thus effective against logic bombs. Archer is implemented using Soot [38] and FlowDroid [4]. Figure 1 gives its architecture.

This section describes two interprocedural analyses. In section 4.2, executor methods are linked to the executee methods they may trigger. In section 4.3, conditional implicit calls are associated with conditions under which they are triggered or ignored. Each of these analyses is formalized in the Inter-procedural Finite Distributive Subset (IFDS) [45] framework.

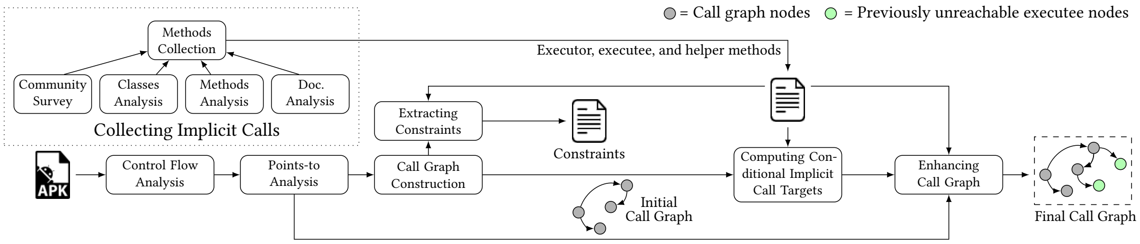


Fig. 1. Overview of our approach to resolving implicit calls and constraints. “Doc.” stands for “Documentation”.

4.1 IFDS framework

The IFDS framework solves context-sensitive interprocedural dataflow problems. It tracks dataflow through a program’s control flow graph, even when the dataflow crosses procedure boundaries. The IFDS framework relies on an *exploded super graph* in which nodes represent abstract dataflow values (from the abstract domain)) at given program statements, and edges represent transfer function behavior.

A node n is a tuple $\langle s, d \rangle$ where s is a program statement and d is a dataflow value from the abstract domain. If a given node $n = \langle s, d \rangle$ is reachable from the initial node $n_0 = \langle s_0, 0 \rangle$, it means the dataflow value d holds before statement s. Here, 0 represents a dummy dataflow value (i.e., it is not in the abstract domain) that always hold, it is used to represent the initial node in the exploded super graph.

Transfer functions can be of four kinds (depicted in Figure 2 of [9]):

(1) Normal edges: model dataflow from a statement that does not contain a procedure call to its successors.
(2) Call edges: model dataflow from call sites to callee methods.
(3) Return edges: model dataflow from return statements to call site receivers.
(4) Call-to-return edges: model intra-procedural dataflow from a statement containing a method call to its successors.

4.2 Resolving the targets of implicit calls

This work proposes a novel approach for resolving the propagation of class literal information to wrapper objects used as arguments to executor methods. A key difference from ordinary constant propagation or dataflow propagation analysis is that our analysis tracks class literals *transitively reachable* from a reference, rather than *only* those literals that might be the value of a particular variable. We demonstrate Algorithm 1, which makes our solution concrete, using the `WorkManager` example from Listing 1.

Call graph construction computes, for each call site, the potential targets or procedure implementations that may be invoked at run time. For object-oriented dispatch, this determines which overriding method implementations might be executed. This can be done using type-based or points-to analysis [53].

For certain implicit calls, the points-to information can identify the potential target of the implicit call. For instance, the `CompletableFuture.runAsync()` method accepts a `Runnable` reference as an argument. The points-to set indicates which potential classes the reference might contain. Together with the executor–executee mapping of section 3, an analysis can deduce which `run()` method will be triggered.

In other cases, the points-to information is not sufficient. For example, in Listing 1, the points-to set for the variable `wr` is of no use in determining the executee for the call to the `enqueue()` method on line 13 with `wr` as the argument. Indeed, a variable of type `WorkRequest` used as a parameter in the `enqueue()` method will have a points-to set that indicates which particular implementation of a given `WorkRequest` is targeted by the variable, but it does not specify which executee will be triggered after the call to the `enqueue()` method. This is because the `WorkRequest` encapsulates a reference to a worker class via class literals, rendering the points-to set ineffective in this context. Standard static analyzers overlook the connection between the `enqueue()` method and the implementation of the `doWork()` method (lines 24–27), so the leak cannot be detected. The same mechanism is used by the Android `JobScheduler` class, which relies on `JobInfo` objects wrapping `ComponentName` objects, which in turn wrap class literals. The following explains how Archer resolves the potential targets of these wrapper objects.

Algorithm 1 Transitively-reachable class literal analysis. The output of the algorithm is a set of edges representing how data flows through the program. An edge $\langle n_i, d_i\rangle \longrightarrow \langle n_{i+1}, d_{i+1}\rangle$ means that dataflow value $d_{i+1} \in \mathcal{P}(C)$ is in the mapping S after statement n_{i+1} if and only if dataflow value d_i is in the mapping S after statement n_i. For brevity, this algorithm omits details of standard call transfer functions such as formal/actual parameters mapping.

```
pathEdges ≔ {$\langle S_0, 0\rangle \longrightarrow \langle S_0, 0\rangle$} // stores edges in the exploded super graph. An edge represents the effect of the dataflow functions and is used by the graph reachability algorithm to check what dataflow values were computed.
workList ≔ {$\langle S_0, 0\rangle \longrightarrow \langle S_0, 0\rangle$} // temporarily stores edges
callToReceiver ≔ ... // a set of methods that we manually determined to propagate dataflow values held by caller object to the variable receiving the returned value of the method call (e.g., $a = b.f()$ would propagate any dataflow value held by $b$ to $a$)
callToBase ≔ ... // a set of methods taking class literals as parameter that we manually determined to generate new dataflow values for caller objects (e.g., $a.f(c)$ would generate a new dataflow value $a \mapsto \{c\}$)
procedure PROPAGATE($n_1, d_1, n_2, d_2$) // propagates dataflow values to the successor statements of a statement to which a flow function has just been applied
    for $s \in succ(n_2)$ do
        edge ≔ $\langle n_1, d_1\rangle \longrightarrow \langle s, d_2\rangle$
        if edge $\notin$ pathEdges then
            Insert edge in pathEdges
            Insert edge in workList
        end if
    end for
end procedure
procedure IFDS( ) // implements transfer functions
    while workList $\neq \emptyset$ do
        Select an edge $\langle n_x, d_x\rangle \longrightarrow \langle n_y, d_y\rangle$ from workList
        switch $n_y$ do
            case $n_y$ is an assignment $a = b$
                if $b$ is a class literal then
                    Propagate($n_x, d_x, n_y, a \mapsto \{b\}$)
                else if $d_y$ is $b \mapsto X$ then
                    propagate($n_x, d_x, n_y, a \mapsto X$)
                end if
            case $n_y$ is an assignment $a = b.f()$
                if $f \in$ callToReceiver and $d_y$ is $b \mapsto X$ then
                    propagate($n_x, d_x, n_y, a \mapsto X$)
                end if
            case $n_y$ is a call statement $a.f(c)$
                if $f \in$ callToBase then
                    if $c$ is a class literal then
                        Propagate($n_x, d_x, n_y, a \mapsto \{c\}$)
                    else if $d_y$ is $c \mapsto X$ then
                        propagate($n_x, d_x, n_y, a \mapsto X$)
                    end if
                end if
            case default
                PROPAGATE($n_x, d_x, n_y, d_y$)
        end switch
    end while
end procedure
```

Archer propagates class literals using the IFDS framework. Each estimated value is a set of class literals in the code of a given app, the abstract domain is the powerset of such literals $\mathcal{P}(C)$, and the lattice is $D = (\mathcal{P}(C), \subseteq)$. Figure 2 illustrates the abstract domain of an app with three class literals. To assign an abstract value to each variable of the program, we use an abstract store: a mapping $S = V \mapsto D$, where V represents the set of variables in the app (fields are

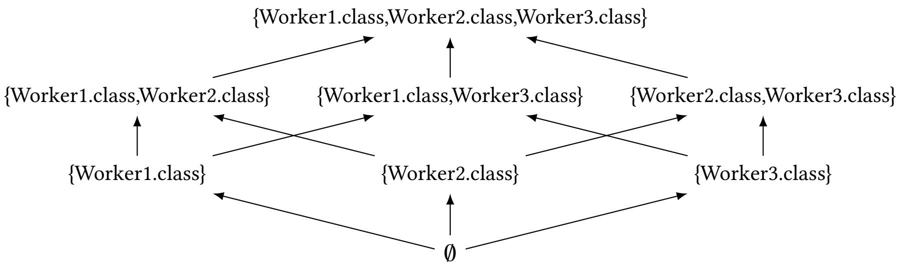
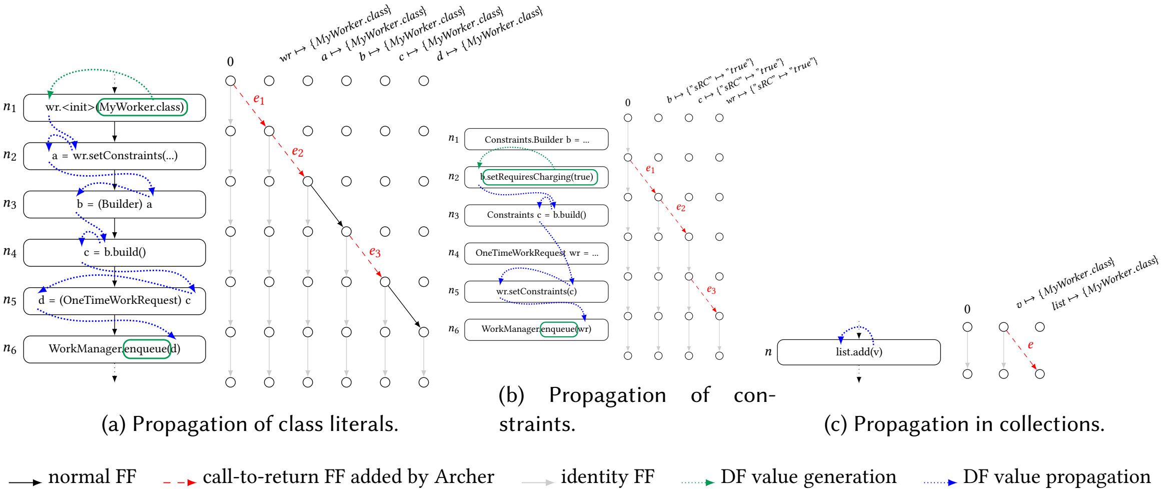

{Worker1.class,Worker2.class,Worker3.class}
{Worker1.class,Worker2.class}
{Worker1.class,Worker3.class}
{Worker2.class,Worker3.class}
{Worker1.class}
{Worker2.class}
{Worker3.class}
∅

Fig. 2. Powerset lattice of three class literals.



(a) Propagation of class literals.

(b) Propagation of constraints.

(c) Propagation in collections.

Fig. 3. How Archer handles call-to-return transfer function within the IFDS framework to generate and propagate dataflow values. Shown is the Jimple representation of code from Listing 1 lines 10–13. To improve comprehensibility, the Jimple code is simplified and the figure omits call and return transfer functions. Standard approaches would not create the red dashed edges in the exploded supergraph. (sRC = setRequiresCharging, FF = flow function, DF = dataflow)

treated equally as variables). For instance, after line 12 of Listing 1, `wr` wraps a reference to `MyWorker.class`, so the dataflow value for variable `wr` would be $\{MyWorker.class\}$.

Archer treats call-to-return transfer functions differently than standard analyses do (see cases 2 & 3 in the switch statement of algorithm 1). A standard analysis uses the identity flow function to propagate dataflow values intra-procedurally after a procedure call (the identity flow function is often used as a default flow function for parts of the control flow graph where the dataflow does not change). When Archer identifies method calls that generate a new mapping for a variable to a dataflow value in the abstract domain (e.g., construction of a new `WorkRequest`: n_1 in Figure 3a), or propagates dataflow values (e.g., setting the constraints to a `WorkRequest`: n_2 in Figure 3a), it not only propagates intra-procedural dataflow values (the job of the identity flow function), but also creates and propagates dataflow values related to implicit calls. For example, when `wr.<init>(MyWorker.class)` is processed by an IFDS solver (in general), the MyWorker.class literals are propagated in the `<init>` method (inter-procedurally). Archer handles this differently by mapping the wr variable to the class literal to retain this information (intra-procedurally).

For instance, consider Figure 3a which depicts part of the code of Listing 1, transformed into Jimple [58], the internal Soot [57] representation. Let us see how a class literal would be propagated by Algorithm 1. In node n_1, in which the `<init>(MyWorker.class)` constructor is called on variable `wr`, a standard transfer function would not change the dataflow value for variable `wr`. However, in our case, we want to know what possible class literal variable `wr` might transitively refer to , possibly through fields, after node n_1. Therefore, our analysis generates dataflow value $wr \mapsto \{MyWorker.class\}$ after calling `<init>(MyWorker.class)` on variable `wr` (edge e_1 on Figure 3a, and lines 34–38 on Algorithm 1). The same reasoning is applied for nodes n_2 and n_4 in which we propagate the dataflow values of variables `wr` and b to variables a and c respectively generating the following dataflow values: $a \mapsto \{MyWorker.class\}$, and $c \mapsto \{MyWorker.class\}$ (cf. lines 27–33 on Algorithm 1). Without doing so, at node n_6, the analysis would never know that variable d refers to class `MyWorker`. This allows our analysis to know that the argument given to the `enqueue()` method is of type `MyWorker`, hence it can correctly connect method `enqueue()` to method `doWork()` of class `MyWorker` thanks to our curated mapping of executor-to-executee. The entire dataflow propagation is depicted in Figure 3a with dotted arrows showing how the dataflow values are generated (from a class literal in the rounded rectangle in node n_1) and propagated to an executor (i.e., rounded rectangle in node n_6).

Handling Collections As explained so far, our strategy would work for method `WorkManager.enqueue(WorkRequest wr)`, but not for method `WorkManager.enqueue(List<WorkRequest> wrs)` whose parameter is a collection of objects. Our novel approach involves adding an extra step in the dataflow analysis, that enables the propagation of dataflow values to collection-like objects, thus allowing to capture the information of the possible references to which the elements of the collection point. To do this, we propagate the dataflow values held by parameters of function calls that allow populating collection-like objects such as lists, sets, etc. Figure 3c depicts this process where in our analysis, variable *list* is mapped to any dataflow values held by variable v. This process allows Archer to know that the collection-like argument given to the `WorkManager.enqueue(List<WorkRequest> wrs)` method holds a reference to class `MyWorker`, hence it can correctly connect method `WorkManager.enqueue(List<WorkRequest> wrs)` to method `doWork()` of class `MyWorker`. Note that in case there are multiple references in the collection, this is not an issue for Archer. Indeed, although our approach is indeed not index-sensitive, the order in which the `WorkRequests` are triggered is managed by the Android framework, hence Archer does not need to retain ordering information. As a result, any downstream analysis need to analyze all executes referenced in the collection.

Enhancing the Call Graph. After collecting the potential targets of executor method calls, we use our previously constructed list of executor–executee method pairs (see Section 3) to retrieve the corresponding executee method for each potential target. For example, in the code shown in Listing 1, if `enqueue()` is the executor method, then `doWork()` is the executee method. We then enhance the call graph with an edge from the executor method to the executee method of the potential target class. This accurately models the implicit method calls in the analyzed Android application.

Soundness of the analysis could be achieved by identifying and adding entry points. Archer's method of adding specific links makes the analysis aware of the real targets, which more precisely represents the application's run-time behavior.

By resolving the target of CI calls, our approach refines the executor–executee links and allows for more precise static analysis of Android apps.

4.3 Extracting constraints

A developer can set execution constraints for CI calls using API calls (e.g., `setRequiresCharging (true)` on line 9 in Listing 1). As described in Section 1, this can be used to trigger logic bombs without using conventional *conditional* statements such as *if* statements. To collect the constraints that trigger an *executee* method, Archer performs an interprocedural dataflow analysis using the IFDS framework, and the following specific configuration. Let M be the set of methods that allow setting a constraint on the execution of an *executee*. Let *Val* be the set of values used to set the actual constraints (e.g., the value `true` on line 9 of Listing 1). Then, the abstract domain is the lattice $D = (\mathcal{P}(M \times Val), \subseteq)$. The abstract store is $S = V \mapsto D$, where V is the set of variables in the app. For instance, in Listing 1, the dataflow value produced after analyzing line 8 for variable `cs` is {`setRequiredNetworkType` ↦ `setCONNECTED`}. After analyzing line 9, the dataflow value for variable `cs` is {`setRequiredNetworkType` ↦ `setCONNECTED`, `setRequiresCharging` ↦ `true`}.

As when resolving CI calls, Archer handles *call-to-return* transfer functions in a non-standard manner: Archer does not only propagate intraprocedural dataflow values. Consider Figure 3b. To determine that certain conditions are set for the `setOneTimeWorkRequest` object passed as argument to the `enqueue()` method (node n_6), Archer propagates the dataflow values to the `Constraint.Builder` object (node n_2 for which Archer generates edge e_1), then to the `Constraint` object (node n_3 for which Archer generates edge e_2), and eventually to the `setOneTimeWorkRequest` object (node n_5 for which Archer generates edge e_3). This computes the conditions that need to be satisfied for an executor method to be executed.

By extracting the constraints that must be met to execute CI calls, our approach provides dynamic analyzers with inputs allowing them to cover code that might not have been covered otherwise. This semi-automated approach (since certain information depends on human identification) improves the coverage and chances to uncover malicious code triggered under circumstances.

5 EVALUATION

This section answers the following research questions:

RQ1: How common are CI calls in Android apps? (§5.1)
RQ2: Does Archer improve static analyzers? (§5.2)
  RQ2.a: What is Archer's effect on benchmark apps?
  RQ2.b: What is Archer's effect on real-world apps?
RQ3: Does Archer enable dynamic analyzers to create inputs that trigger CI calls? (§5.3)
  RQ3.a: What is Archer's effect on benchmark apps?
  RQ3.b: What is Archer's effect on real-world apps?

Datasets: To evaluate Archer, we created two datasets:

- **benchmark_dataset:** We followed the methodology used for the popular DroidBench benchmark [4]: we created 16 new apps that use various executor, executee, and helper classes and methods, as well as constraints to trigger executees under specific circumstances (see our replication package for details). We intentionally inserted data leaks into 12 of these apps, while the remaining 4 apps were left without leaks to assess false positives. The average size of these apps is 3.1 MB and the standard deviation is 0.05 MB.
- **real_world_dataset:** We randomly selected 3000 goodware and 3000 malware from AndroZoo [2] (AndroZoo contains 24 million apps at the time of writing). An app is considered a goodware if none of the 60+ antiviruses from VirusTotal [28] flags it. An app is considered malware if at least 5 of the 60+ antiviruses from VirusTotal flag it. We considered apps for which the dex file date was 2020 or later (i.e., recent apps compatible with Android API level 30). The average size and the standard deviation for malware are 32.5 MB and 31.4 MB, and for goodware it is 37.8 MB and 34.6 MB. These values are within 4% of the respective values for all of AndroZoo.

	Total	Executor	Executee	Helper	None
Goodware	3000 (100%)	576 (19%)	889 (30%)	212 (7%)	2111 (70%)
Malware	3000 (100%)	350 (12%)	447 (15%)	225 (8%)	2548 (85%)

Table 2. Number of apps that use CI calls.

	# in goodware	# in malware
Total	163 945 (100%)	38 609 (100%)
`Runnable`	149 315 (91.1%)	36 368 (94.2%)
`Callable`	11 855 (7.2%)	998 (2.6%)
`TimerTask`	2348 (1.4%)	593 (1.5%)
`JobService`	298 (0.2%)	636 (1.6%)
`Worker`	124 (0.1%)	14 (0.1%)
`RunnableScheduledFuture`	5 (0%)	0 (0%)

Table 3. Occurrences of executees in our **real_world_dataset**.

5.1 RQ1: How common are CI calls in real-world Android apps?

This section presents a quantitative analysis of CI call usage in our **real_world_dataset**.

Results. Table 2 gives the number of apps that utilize executors, executees, and helpers. This Table shows that a higher proportion of goodware employs CI calls. Furthermore, executees are more commonly used than executors and helpers in goodware and malware. There are two reasons for this. First, an executor can trigger multiple executees, and the executees studied in this paper include the widely-used `Runnable` class, which is often used to initiate `Threads` in apps. Second, a use of an executee might not always be related to a CI call (but note that a use of an executor is always related to a CI call). Table 3 shows the occurrences of different executees, with `Runnable` by far the most frequently used in goodware and malware.

Figure 4 shows the distribution of the numbers of occurrences of executors, executees, and helper methods in our **real_world_dataset**. Goodware tends to have more executors per app, similar executees, and slightly fewer helpers. These results are based on apps that have at least one executor, executee, or helper, respectively. Otherwise, the median would always be 0.

Among the 576 goodware apps containing executors, 453 have executors that are reachable in the call graph generated by the CHA call graph construction algorithm (i.e., they are directly called by a method that is in the call graph). Additionally, 707 apps have reachable executees, and 187 have reachable helpers in the call graph. For the malware apps, 284 have reachable executors in the call graph, 250 have reachable executees, and 222 have reachable helpers.

Among the 576 goodware apps that call executors, 441 (76.6%) rely on both temporal and conditional triggers, 55 (9.5%) rely only on conditional triggers (i.e., no temporal constraint), and 80 (13.9%) rely solely on temporal constraints. Regarding the 350 malware that call executors, 266 (76%) rely on both temporal and conditional triggers, 22 (6.3%) rely only on conditional triggers (i.e., no temporal constraint), and 62 (17.7%) rely solely on temporal constraints.

RQ1 answer: Conditional implicit calls can be found in 20% of Android goodware apps, and in 12% of malware.

5.2 RQ2: Archer's effect on static analyzers

This section evaluates the performance of Archer in improving the soundness and precision of static analysis tools. That is, Archer adds missing edges to Android apps' call graphs and resolves CI call targets that would otherwise be left unanalyzed.

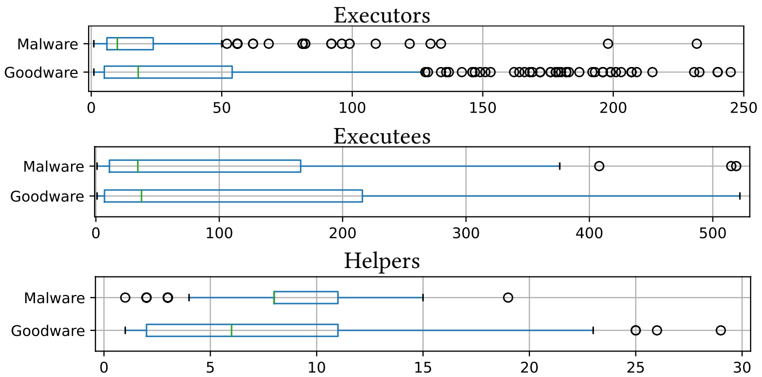


Fig. 4. Distribution of the number of occurrences of executors, executees, and helpers in our **real_world_dataset**, excluding “0” datapoints.

⊛ = true-positive, ★ = false positive, ◯ = false negative, ● = leak, ○ = no leak

Test Case	Leak	T	C	FlowDroid	IccTA	RAICC	Amandroid	DroidSafe	DroidRA	Archer
WorkManager_enqueue	●		●	◯	◯	◯	◯	◯	◯	⊛
WorkManager_enqueueUniqueWork	●		●	◯	◯	◯	◯	◯	◯	⊛
WorkManager_enqueue1	○		●							
TimerTask_schedule	●	●		⊛	◯	⊛	◯	◯	⊛	⊛
JobScheduler_schedule	●	●	●	⊛	◯	⊛	◯	◯	⊛	⊛
JobScheduler_schedule1	○			★		★			★	
CompletableFuture_runAsync	●			◯	◯	◯	◯	◯	◯	⊛
CompletableFuture_thenRun	●			◯	◯	◯	◯	◯	◯	⊛
CompletableFuture_runAsync1	○									
ExecutorCompletionService_submit	●			◯	◯	◯	◯	◯	◯	⊛
STPE_scheduledAtFixedRate	●	●		◯	◯	◯	◯	◯	◯	⊛
STPE_invokeAll	●			◯	◯	◯	◯	◯	◯	⊛
STPE_scheduleWithFixedDelay	○	●								★
ExecutorService_submit	●			◯	◯	◯	◯	◯	◯	⊛
PE_scheduleWithFixedDelay_enqueue	●	●		◯	◯	◯	◯	◯	◯	⊛
SynchronousExecutor_execute	●			◯	◯	◯	◯	◯	◯	◯
Overall										
Count of ⊛, higher is better				2	0	2	0	0	2	11
Count of ★, lower is better				1	0	1	0	0	1	1
Count of ◯, lower is better				10	12	10	12	12	10	1
Precision p = ⊛/(⊛+ ★)				67%	0%	67%	0%	0%	67%	92%
Recall r = ⊛/(⊛+ ◯)				17%	0%	17%	0%	0%	17%	92%
F_1-score = $2pr/(p+r)$				27%	0%	27%	0%	0%	27%	92%

Table 4. Data leak detection on **benchmark_dataset** (T = temporal trigger, C = conditional trigger).

5.2.1 *RQ2.a: Archer’s effect on benchmark apps.*

To evaluate the effectiveness of Archer in resolving CI calls, we compare it to state-of-the-art dataflow analyzers, as commonly performed in the literature [36, 39, 48, 50, 55, 61]. We only include publicly-available analyzers that account for (some) implicit calls: FlowDroid [4], IccTA [36], RAICC [48], Amandroid [61], DroidSafe [29], and DroidRA [37].

Results. Table 4 summarizes the results of this experiment. The six tools perform poorly on this dataset: they overlook CI calls. FlowDroid (and RAICC and DroidRA, which rely on FlowDroid for taint analysis) did report three leaks (of which one is a false positive), but these were due to specific cases that are hard-coded or over-approximated by FlowDroid. For example, in the sample `TimerTask_schedule`, FlowDroid implements a hard-coded heuristic for this particular case

	Before Archer		**After Archer**		**Difference**	
	# Nodes	# Edges	# Nodes	# Edges	Added Nodes	Added Edges
Goodware	8188	41 723	8471	42 425	283 (+3%)	702 (+2%)
Malware	8291	33 545	8652	34 548	361 (+4%)	1003 (+3%)

Table 5. Average number of nodes and edges per app.

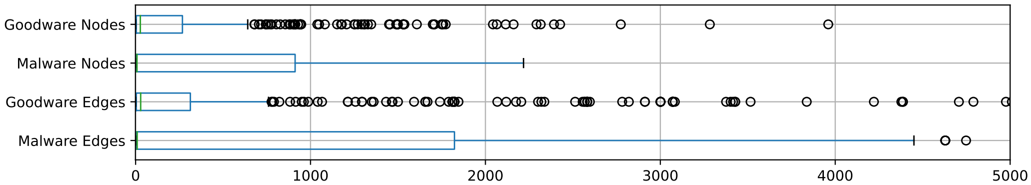


Fig. 5. Distribution of new call graph nodes and edges.

(see [5], lines 564–647). In the sample `JobScheduler_schedule`, the `JobService` executee class is an Android Service component, and FlowDroid considers its methods inherited by Android framework classes (i.e., `JobService`) as potential callbacks, leading to an over-approximation. As a result, the code in method `onStartJob()` of class `JobService` is covered by FlowDroid. The third reported leak in sample `JobScheduler_schedule1` is a false-positive result, as method `onStartJob()` is not actually triggered using methods such as `schedule()` in this sample. Since FlowDroid models method `onStartJob()` as potentially called, it incorrectly reports a leak. IccTA, Amandroid, and DroidSafe obtained a zero score. These tools do not resolve the CI calls studied in this paper, so they do not reach and analyze the code written in executees.

Archer performs best: its precision, recall, and f_1 score are 92%. For sample `JobScheduler_schedule1`, no false positive is reported because Archer does not apply FlowDroid's over-approximation for the `onStartJob()` method. Archer does issue a false positive for sample `ScheduledThreadPoolExecutor_scheduleWithFixedDelay`, in which there is no leak at run time because the constraints under which the executee should be executed will never be met. Indeed, the scheduleWithFixedDelay() method is called with the argument "−1" for the delay, and a delay value of -1 will trigger an exception, preventing the Runnable from being executed. However, Archer does not statically check if the constraints are realizable, resulting in an over-approximation. Archer suffers a false-negative for sample `SynchronousExecutor_execute`, where the executee is triggered using reflection, which Archer does not handle.

RQ2.a answer: Archer outperforms state-of-the-art static dataflow analyzers in detecting CI calls in Android apps. On a benchmark, Archer achieves an F_1 score of 92%, while FlowDroid, RAICC, and DroidRA achieve an F_1 score of 17%, and Amandroid, IccTA, and DroidSafe achieve an F_1 score of 0%. This reveals Archer's ability to detect logic bombs in Android apps, i.e., it statically detects data leaks triggered under certain circumstances.

5.2.2 RQ2.b: Archer's effect on real-world apps. We measured the extent to which Archer augments apps' call graphs. We used apps calling executor methods from our **real_world_dataset**, i.e., 576 goodware and 350 malware (see Table 2).

Table 5 reports the number of nodes and edges in call graphs before (i.e., FlowDroid) and after applying Archer on our dataset. A call graph node represents a method and an edge represents the calling relationship between two methods. There are more edges than nodes on average since a node can be the target of multiple edges. Figure 5 shows the distribution of the number of new call graph nodes and edges introduced by Archer.

Figure 6 plots the distribution of the number of additional Jimple statements [58] reachable thanks to Archer. We computed the number of additional statement to show that missing "nodes" in the call graph, leads to potentially

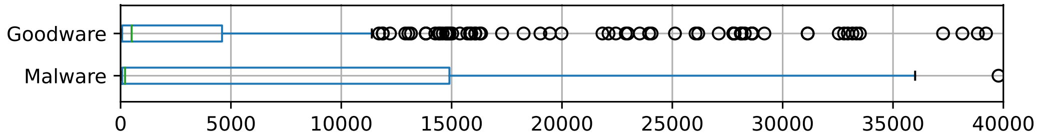


Fig. 6. Distribution of extra reachable Jimple statements.

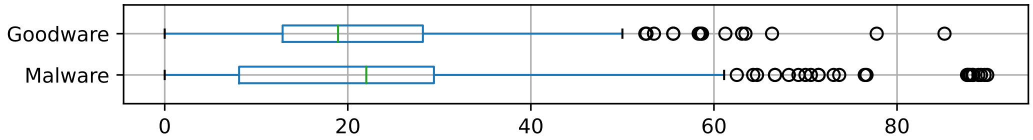


Fig. 7. Distribution of the time overhead introduced by Archer to resolve CI calls (in %). The overhead is the extra time Archer needs to resolve CI calls.

missing a substantial part of the code. The average number of extra statements that are reachable for goodware is 5260, and the median is 499. The average number of extra statements that are reachable for malware is 6206, and the median is 205.

These numbers indicate that static analyzers that do not model CI calls will miss a significant number of methods in each app [39]. In both goodware and malware apps, more than 3% of nodes are added by Archer, meaning that existing tools would overlook these methods which is unacceptable from a security point of view (note that the previously discussed heuristic of FlowDroid in Section 5.2.1 would only help covering 3% of the edges brought by Archer in goodware and 7% in malware). Indeed, this can be particularly problematic in the case of malware detection, as it may allow malware to evade detection if the malicious code is hidden in a CI call and not detected by static analyzers. This could result in the app being mistakenly considered safe and allowed into the Google Play [32].

Analysis time overhead: Archer is built on top of FlowDroid and requires additional steps to resolve CI calls. We measured the time taken by Archer to resolve CI calls and the time that Soot/FlowDroid takes to load an app. Figure 7 shows that Archer introduces an overhead of about 20% (21 extra seconds when loading malware and 40 extra seconds for goodware, on average). We consider the benefit of plugging soundness holes to justify this overhead.

Manual analysis To assess whether the edges added are correct and go further than recent literature [39], we manually analyzed Archer's output.

Among the 696 207 call graph edges (330 943 in goodware + 365 264 in malware), from an executor to an executee, added by Archer in both goodware and malware, we randomly chose 100 edges (confidence level of 95% and confidence interval of ± 10%). For every edge (*executor* → *executee*), we followed this process: ① we downloaded the app from AndroZoo [2]; ② we decompiled the app using Jadx [52] to get readable source code representation of the Dalvik bytecode embedded in the app; ③ we manually analyzed the method in which *executor* is called to check the argument used to trigger it; and ④ if the argument used to trigger the executor is related to *executee*, we mark it as correct, incorrect otherwise. For instance, suppose *executor* is `enqueue()`. If *executee* is `MyWorker.doWork()`, then we mark the edge *executor* → *executee* as a true positive, otherwise a false positive. As a concrete example, consider Listing 1. If Archer yields an edge such as $WorkManager.enqueue() \longrightarrow MyWorker.doWork()$, we manually analyze method `MainActivity.onCreate()` (since `enqueue()` is called in `MainActivity.onCreate()`) and check the argument(s) used to trigger it. In this case, the argument is *wr*, which holds a reference to class `MyWorker` (lines 12–13). Hence the edge is marked as a true positive.

Overall, we found that 88/100 of the edges analyzed are correct (precision 88%). All of the 12 incorrect edges were due to the over-approximation of the SPARK [35] points-to algorithm to infer targets of CI calls (see Section 4.2). This matter is further discussed in Section 6.

Archer's ability to detect new leaks. Given that Archer can access a greater number of nodes, it is anticipated to identify additional data leaks. This section checks whether Archer truly enhances the detection of data leaks in real-world apps, a capability already demonstrated in benchmark apps. To do so, as in [39], we applied both FlowDroid before and after Archer (with the same list of sources and sinks) on apps calling executor methods from our **real_world_dataset**, i.e., 576 goodware and 350 malware (see Table 2). Archer enhanced the capability of FlowDroid, leading to the discovery of 68 additional data leaks in the 350 malware apps and 177 additional data leaks in the 576 goodware apps. The authors have manually checked 50 data leaks (representative sample with a confidence level of 90% and a margin of error of 10%) among the 177 additional data leaks in goodware and reported that only two were false positives (i.e., the authors were not able to manually find the data flow path provided by Flowdroid in the given apps). Tables 6 and 7 show the top 5 sources and sinks in the newly found data leaks thanks to Archer.

Table 6. Top 5 sources and sinks in additional leaks in malware found by Archer.

Sources	
Occurrence	Method
54	android.database.Cursor.getString(int)
4	android.net.wifi.WifiInfo.getSSID()
3	android.telephony.gsm.GsmCellLocation.getCid()
3	android.telephony.gsm.GsmCellLocation.getLac()
1	android.telephony.TelephonyManager.getDeviceId()
Sinks	
Occurrence	Method
21	android.content.IntentFilter.addAction(String)
14	android.os.Bundle.putParcelable(String,Parcelable)
10	android.content.Context.registerReceiver(BroadcastReceiver,IntentFilter)
7	android.content.Context.bindService(Intent,ServiceConnection,int)
7	android.os.Bundle.putSparseParcelableArray(String,SparseArray)

Table 7. Top 5 sources and sinks in additional leaks in goodware found by Archer.

Sources	
Occurrence	Method
89	android.telephony.TelephonyManager.getDeviceId()
28	android.location.LocationManager.getLastKnownLocation(String)
25	android.location.Location.getLatitude()
17	android.location.Location.getLongitude()
11	android.database.Cursor.getString(int)
Sinks	
Occurrence	Method
90	android.content.SharedPreferences$Editor.putString(String,String)
19	android.util.Log.d(String,String)
18	android.content.Context.registerReceiver(BroadcastReceiver,IntentFilter)
10	android.util.Log.e(String,String,Throwable)
10	android.os.Bundle.putAll(Bundle)

⍟ = true-positive, ◯ = true-negative
● = constraint, ○ = no constraint

Test Case	# Constraints	T	C	# Detection
WorkManager_enqueue	●●		●	⍟⍟
WorkManager_enqueueUniqueWork	● ● ●		●	⍟ ⍟ ⍟
WorkManager_enqueue1	●		●	⍟
TimerTask_schedule	●	●		⍟
JobScheduler_schedule	● ● ●	●	●	⍟ ⍟ ⍟
JobScheduler_schedule1	○			◯
CompletableFuture_runAsync	○			◯
CompletableFuture_thenRun	○			◯
CompletableFuture_runAsync1	○			◯
ExecutorCompletionService_submit	○			◯
ScheduledThreadPoolExecutor_scheduledAtFixedRate	● ● ●	●		⍟ ⍟ ⍟
ScheduledThreadPoolExecutor_invokeAll	○			◯
ScheduledThreadPoolExecutor_scheduleWithFixedDelay	● ● ●	●		⍟ ⍟ ⍟
ExecutorService_submit	○			◯
Poolexecutor_scheduleWithFixedDelay_enqueue	● ● ●	●		⍟ ⍟ ⍟
SynchronousExecutor_execute	○			◯

Table 8. Results of constraints detection on our benchmark (T = temporal trigger, C = conditional trigger).

RQ2.b answer: Archer discovers previously unreachable methods (~4% on average) for inter-procedural dataflow analyses. Archer's precision is 88%, and the imprecision is due to the SPARK points-to analysis used in our experiments. Archer makes FlowDroid detect previously unknown data leaks in real-world apps.

5.3 RQ3: Archer's effect on dynamic analyzers

This section evaluates Archer's ability to extract the conditions under which CI calls are executed. If Archer can provide dynamic analyzers with inputs to cover these CI calls, it will improve the coverage of dynamic analysis for Android apps, thereby improving the chances to discover logic bombs.

5.3.1 RQ3.a: Archer's effect on benchmark apps. On our **benchmark_dataset**, Archer achieves a perfect score (Table 8). For apps without constraints, Archer does not report any constraints. For apps with constraints, Archer accurately reports all the constraints with the correct parameters. For example, in sample `WorkManager_enqueueUniqueWork`, the implicit call is only triggered if the device is not in an idle state, is connected to the Internet, and is charging.
Archer's benefit to dynamic analysis. To approximate how much Archer's constraints aid dynamic analyzers, we measured code coverage. We used ACVTool [44] to monitor apps' execution in an emulator with ① default emulator configuration; and ② emulator configured with the constraints provided by Archer (e.g., device in charge or not). In both cases, we use the same inputs generated with Google's Monkey [27].

We selected apps with constraints in the code (see Table 8) and code executed in the executee for this experiment. Table 9 shows that Archer's constraints increase code coverage by 17 percentage points on average. We manually confirmed that executee methods were not triggered when the conditions were not set properly.

RQ3.a answer: On our benchmark dataset, Archer extracts all the conditions under which implicit calls can occur. Archer's extracted conditions improve the performance of dynamic analyzers by increasing code coverage. In other words, malicious behavior triggered under certain circumstances (i.e., logic bombs) with CI calls can be reached thanks to Archer.

	With Archer Constraints	With default Constraints
WorkManager_enqueue	80%	65%
WorkManager_enqueueUniqueWork	83%	71%
WorkManager_enqueue1	78%	59%
TimerTask_schedule	70%	56%
JobScheduler_schedule	78%	63%
ScheduledThreadPoolExecutor_scheduledAtFixedRate	73%	60%
Poolexecutor_scheduleWithFixedDelay_enqueue	74%	45%
Average	77%	60%

Table 9. Code coverage of the main app package (i.e., not library code) with and without constraints satisfied in the execution environment.

5.3.2 RQ3.b: Archer's effect on real-world apps. This section evaluates the precision of the constraints generated by Archer that must be met in order to trigger CI calls. To do so, we executed Archer on apps having calls to executor methods in our **real_world_dataset**, i.e., 576 goodware and 350 malware (Table 2).

Just under half of these apps use constraints when using executors: 253/576 (44%) for goodware and 144/350 (41%) for malware. We found 820 constraints in goodware and 327 in malware. Figure 8 reports the distribution of the number of constraints found per app.

Most of the constraints we observed are temporal constraints (based on the list we previously built, see Section 3). Indeed, 68% of constraints in goodware require triggering the executee after a delay and 59% in malware. Other constraints involved in goodware are: run executee periodically (3%), the device is connected to a specific network type (3%), the device is charging (2%), the device is in an idle state (1%), executee is persistent (5%), the time unit extracted to set time-related constraints (18%). Other constraints involved in malware are: the device is charging (2%), the device is connected to a specific network type (2%), executee is persistent (31%), the device is in an idle state (1%), run executee periodically (0.3%), the time unit extracted to set time-related constraints (4%).

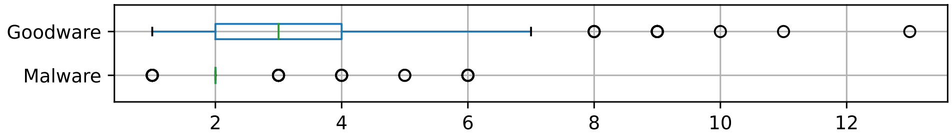


Fig. 8. Distribution of the number of constraints per app. For malware, the median is 2.

We conclude that: ① dynamic analyzers might not cover part of the code in almost half of the apps if conditions to trigger implicit calls are not met; and ② several constraints are found per app that condition the triggering of executees which can complicate the tasks of dynamic analyzers to cover the code triggered.

Manual analysis. Since static analyzers often suffer from false-positive results, the authors of this paper manually analyzed Archer's results. We randomly sampled 100 of 1147 (820 in goodware + 327 in malware) constraints found by Archer, providing a confidence level of 95% and confidence interval of ± 10%. For each sample, we ① downloaded the app from AndroZoo; ② decompiled the app using Jadx; ③ read the method in which the executor is called; and ④ if the constraints found by Archer (e.g., trigger after 10 seconds) correspond to the code decompiled, we mark it as correct, incorrect otherwise. For instance, for Listing 1, Archer yields the following constraints: ① the device is connected to a network; and ② the device is charging, to execute the `Worker.doWork()` method. We would manually analyze method `MainActivity.onCreate()` and check what are the constraints set to trigger the executee. In this case, the constraints yielded by Archer are marked as correct.

The constraints found by Archer were correct in 96 cases out of 100 (96%). For the remaining 4 apps, the types of the constraints were correct (e.g., a specific time before execution). However, Archer could not statically determine the values used to set the constraints, since the app computed them at run time.

The Scylla malware. We also assessed Archer's ability to help dynamic analyzers on a real-world Android malware that uses CI calls, known as Scylla [32] which entered Google Play. The Scylla malware uses the `JobScheduler` [32] to execute its malicious code under specific conditions. This allows Scylla to operate undetected, making it a subtle yet effective security threat. To better explain, let us focus on a specific instance of the Scylla malware[1]. In function useJobServiceForKeepAlive() of Service com.Talentnewdev.gjxFfCOcq.KbLDfknMveZUxWuT, the Jobscheduler is used to trigger a JobService (class com.Talentnewdev.gjxFfCOcq.ScheduleService). When its onStartJob() method is triggered, it checks whether Service com.Talentnewdev.gjxFfCOcq.KbLDfknMveZUxWuT is alive or not, in case it is not, it launches it, creating an infinite loop that would trigger Service com.Talentnewdev.gjxFfCOcq.KbLDfknMveZUxWuT forever. When the latter is created and executed, code to check whether the screen is on or off is executed. If the screen is off, as described in [32], a fake window is displayed (i.e., with a size of 0), in our example, using the com.Talentnewdev.gjxFfCOcq.TekuHhrsAGActivity class, to show ads. Then the app will generate fake clicks to generate revenue. Eventually method useJobServiceForKeepAlive() is triggered again to restart the cycle. This example illustrate that if the code triggered by the JobScheduler is not executed at run time because the conditions are not met, then a dynamic analyzer might never cover the code and leave the malware undetected (Scylla entered Google Play).

Similar to the previous section, we ran the Scylla malware with and without the constraints extracted by Archer. Since the constraints were correclty set, the malicious code (well described at https://www.humansecurity.com/learn/blog/poseidons-offspring-charybdis-and-scylla) was successfully triggered (i.e., the code was covered by ACVTool). This result suggests that dynamic analyzers, designed for identifying malicious activities, have better chances of detecting malware when supported by Archer (i.e., with additional inputs given by Archer). The interested reader can refer to the README file of Archer's GitHub repository in which we give the execution result of Archer on the given Scylla sample.

RQ3.b answer: Almost half of the apps using CI calls rely on constraints to execute the code, challenging dynamic analyzers to cover parts of the code. Archer can precisely report constraints used to trigger CI calls in 96% of the cases. Thus, Archer helps dynamic analyzers to uncover malicious behavior.

6 LIMITATIONS AND THREATS TO VALIDITY

We manually analyzed the Android documentation to collect ways to perform CI calls in the Android framework. Although we followed a systematic approach, it is possible that we missed some mechanisms for performing CI implicit calls in the Android framework, though our process allows for high precision. We mitigate this threat to validity in two ways: we make all our artifacts available to the community for further verification and exploration, and if new CI call mechanisms are discovered, they can easily be integrated into Archer.

Archer accounts for implicit calls which can be *explicitly* constrained, i.e., it does not account for calls that are triggered under certain circumstances such as the `onPause()` method. The rationale behind our study is to focus on potential malicious trigger-based behavior, in which conditions have to be explicitly set by malicious developers (with a reasoned choice).

Archer over-approximates the behavior of certain CI calls. In cases where a CI call mechanism cannot execute a particular executee at run time because the constraints will never be met (e.g., triggering an executee in the past, as

[1] SHA-256: 2D46A76C94DFF80992615354C713E1552A1F55F3EB0C4D5297D52571180D6402

discussed in Section 5.2.1), Archer still connects the executor and potential executees involved. Future work could check if the constraint is feasible at run time to improve the accuracy of Archer.

Archer uses points-to analysis to infer the potential targets of executor methods. As a result, it shares the limitations of the algorithm used to compute the points-to set of variables, such as over-approximation of the targets.

7 RELATED WORK

This section discusses other research on resolving specific kinds of implicit control flow. Our study is the first to catalog and analyze general Android framework mechanisms for conditional implicit calls (i.e., triggering code under specific circumstances). None of the previous work addresses these types of triggers, and a tool should incorporate both previous techniques and our new contributions to fully capture implicit control flow in Android apps.

Inter-Component Communication. Android apps are made of *components* communicating through inter-component communication (ICC) [40]. ICC methods provided by the Android framework (e.g., `startActivity()`, `startService()`) [4] trigger (i.e., implicitly call) *lifecycle methods* (e.g., `onCreate()`, `onBind()`).

A large body of work tries to resolve target components of ICC communication. IccTA [36] infers `Intent` potential targets using IC3 [41]. Amandroid [61] infers possible target components by generating a component-wise dataflow graph and component-level data dependence graph. Amandroid's engine then relies on a summary table that models ICC channels. DroidSafe [29] relies on string and class designator analysis to infer potential target components, then DroidSafe modifies ICC method calls into explicit lifecycle method calls. RAICC [48] resolves "atypical" ICC methods (e.g., `AlarmManager.set()`). "Atypical" methods usually rely on `PendingIntent` or `IntentSender` objects wrapping component targets. After resolving potential targets with new IC3 rules, RAICC instruments the app and adds typical ICC method (e.g., `startActivity()`) calls. ICCBot [64] infers the component transition (i.e., ICC) that are connected via Android's fragments (i.e., modular and reusable components of an activity that represents a portion of a GUI) [12]. ICCBot performs a context-sensitive and inter-procedural analysis to precisely model data carried by ICC objects (e.g., Intents). Chen et al. [13, 14] developed an approach to construct an Activity Transition Graph (ATG) to build Android apps' storyboards. To do so, the authors rely on ICC-related information used to trigger new Activities.

Callbacks. Wu et al. [62] proposed a callback-aware approach to detect resource leaks in Android apps. The authors focus on two types of callback methods: system-triggered callbacks and user-triggered callbacks. The former represents, in their study, lifecycle methods and resource classes' callback methods (e.g., `onPause()`) while the latter represents callbacks triggered by user interaction with the GUI. Similarly, Yang et al. [65] study lifecycle and user-driven callbacks. Their approach relies on a GUI model by generating a callback control flow graph. It then extracts possible sequences of user GUI events derived from valid paths in the GUI model.

EdgeMiner [11] statically finds implicit control flow through the Android framework. The authors focus on the registration mechanisms that allow passing procedures as parameters. A classic example is the `setOnClickListener()` method that triggers the `onClick()` method. This approach aims to identify as many implicit calls as possible within the framework. The authors produce a list of pairs that represents the potential executors and executees of implicit calls. Contrary to our work, their list is not manually vetted, so it is possible that some of the pairs may not represent actual implicit calls. Additionally, they do not present a method for resolving the potential targets of these calls or for accounting for the conditions that may trigger them. Yang et al. [66] proposed an approach based on the effect of GUI-related callbacks to construct a model of the behavior of Android apps' user interfaces.

Reflection. Reflection permits *introspection* at execution time. For instance, `method.invoke(object)` triggers the execution of the method represented by `method` on `object`, but the method's name is not a token in the source code.

DroidRA [37, 55] is an instrumentation-based analysis tool that modifies Android apps to make them more amenable to static analysis by making implicit calls explicit. The authors resolve reflective calls using the COAL [41] solver to infer reflection targets. They instrument the app for analysis purpose, and for each reflective call resolved, a corresponding explicit call is added in the app. Besides `Intent` resolution, SPARTA [7] resolves Java reflection calls targets. Their solution is two-fold: ① a reflection type system tracking and inferring potential names of classes and methods; ② a reflection solver estimating method signatures that can be invoked. Blackshear et al. [8] proposed Droidel, an approach to make the Android framework more amenable to static analysis by manually replacing reflective calls from the Android framework with direct calls.

Qualitative Analyses. Pan et al. [42] identified five techniques to perform asynchronous tasks. They implement AsyncChecker and conduct a qualitative analysis to check for misuse in Android apps. You et al. [67] study the possible implicit information flow that can arise in Android's Dalvik bytecode. They develop a control-transfer-oriented analysis in a formal structured semantic model. They show which Dalvik instructions are responsible for implicit information flow: `if`, `switch`, `throw`, etc. Fengguo et al. [60] explore how Android malware uses task scheduling to trigger malicious code. For instance, they discovered that malware relies on recurring tasks with `Thread` objects to receive commands from external servers. Linghui et al. [39] recently introduced GenCG, an approach designed to simplify the modeling of Java frameworks, including the Android framework. They conduct a qualitative evaluation of GenCG on benchmark apps, demonstrating that the application of their tool results in more static code coverage. Contrary to these approaches, we do not aim to provide qualitative information about the implementation of mechanisms. We propose to improve state-of-the-art static analyzers with previously overlooked links between method calls that can be used to trigger code under specific circumstances.

8 CONCLUSION

Conditional implicit calls are common in Android apps, and they present a challenge for both static and dynamic analysis. We developed Archer, a tool that improves the soundness and precision of static analysis for Android apps. Archer does this by adding new edges to the call graph that were previously overlooked, as well as by resolving the targets of CI calls. In addition, Archer provides dynamic analyzers with the conditions under which these calls may be executed by extracting them from the code. Our evaluation of Archer demonstrates its effectiveness at improving the coverage and accuracy of both static and dynamic analysis for Android apps. Overall, our work contributes to the research effort in understanding and analyzing implicit calls in the Android framework as well as making the Android ecosystem a safer place for end-users.

9 DATA AVAILABILITY

To promote transparency and facilitate reproducibility, we are making the artifacts used in our study available to the community. This includes the source code and executable of Archer, the datasets used in our experimentation, our scripts to run third-party tools, our benchmark apps, the results produced, and any other artifacts related to our study.

We archived our artifacts on the preserved repository *Zenodo*:

https://github.com/JordanSamhi/Archer
https://doi.org/10.5281/zenodo.10474310

REFERENCES

[1] Allen, F. E. Control flow analysis. *SIGPLAN Not. 5*, 7 (July 1970), 1–19.

[2] Allix, K., Bissyandé, T. F., Klein, J., and Le Traon, Y. AndroZoo: Collecting millions of Android apps for the research community. In *Proceedings of the 13th International Conference on Mining Software Repositories* (New York, NY, USA, 2016), MSR '16, ACM, pp. 468–471.

[3] Andersen, L. O. *Program analysis and specialization for the C programming language.* PhD thesis, Cornell, 1994.

[4] Arzt, S., Rasthofer, S., Fritz, C., Bodden, E., Bartel, A., Klein, J., Le Traon, Y., Octeau, D., and McDaniel, P. FlowDroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. *SIGPLAN Not. 49*, 6 (June 2014), 259–269.

[5] Arzt, S., Rasthofer, S., Fritz, C., Bodden, E., Bartel, A., Klein, J., Le Traon, Y., Octeau, D., and McDaniel, P. Flowdroid source code, https://github.com/secure-software-engineering/FlowDroid/blob/develop/soot-infoflow/src/soot/jimple/infoflow/cfg/LibraryClassPatcher.java, 2022. Accessed April 2022.

[6] Bacon, D. F., and Sweeney, P. F. Fast static analysis of c++ virtual function calls. *SIGPLAN Not. 31*, 10 (Oct. 1996), 324–341.

[7] Barros, P., Just, R., Millstein, S., Vines, P., Dietl, W., d'Amorim, M., and Ernst, M. D. Static analysis of implicit control flow: Resolving Java reflection and Android intents. In *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering* (2015), ASE '15, p. 669–679.

[8] Blackshear, S., Gendreau, A., and Chang, B.-Y. E. Droidel: A general approach to android framework modeling. In *Proceedings of the 4th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis* (New York, NY, USA, 2015), SOAP 2015, Association for Computing Machinery, p. 19–25.

[9] Bodden, E. 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* (2012), SOAP '12, p. 3–8.

[10] Cao, M., Ahmed, K., and Rubin, J. Rotten apples spoil the bunch: An anatomy of google play malware. In *Proceedings of the 44th International Conference on Software Engineering* (2022), ICSE '22, p. 1919–1931.

[11] Cao, Y., Fratantonio, Y., Bianchi, A., Egele, M., Kruegel, C., Vigna, G., and Chen, Y. EdgeMiner: Automatically detecting implicit control flow transitions through the Android framework. In *NDSS* (2015).

[12] Chen, J., Han, G., Guo, S., and Diao, W. FragDroid: Automated user interface interaction with activity and fragment analysis in Android applications. In *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)* (2018), pp. 398–409.

[13] Chen, S., Fan, L., Chen, C., and Liu, Y. Automatically distilling storyboard with rich features for Android apps. *IEEE Transactions on Software Engineering* (2022).

[14] Chen, S., Fan, L., Chen, C., Su, T., Li, W., Liu, Y., and Xu, L. Storydroid: Automated generation of storyboard for Android apps. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)* (2019), pp. 596–607.

[15] Choudhary, M., and Kishore, B. Haamd: Hybrid analysis for Android malware detection. In *2018 International Conference on Computer Communication and Informatics (ICCCI)* (2018), pp. 1–4.

[16] Dean, J., Grove, D., and Chambers, C. Optimization of object-oriented programs using static class hierarchy analysis. In *ECOOP'95 — Object-Oriented Programming, 9th European Conference, Åarhus, Denmark, August 7–11, 1995* (Berlin, Heidelberg, 1995), Springer Berlin Heidelberg, pp. 77–101.

[17] Ernst, M. D., Just, R., Millstein, S., Dietl, W., Pernsteiner, S., Roesner, F., Koscher, K., Barros, P. B., Bhoraskar, R., Han, S., Vines, P., and Wu, E. X. Collaborative verification of information flow for a high-assurance app store. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security* (2014), CCS '14, p. 1092–1104.

[18] Fereidooni, H., Conti, M., Yao, D., and Sperduti, A. ANASTASIA: ANdroid mAlware detection using STatic analySIs of Applications. In *2016 8th IFIP International Conference on New Technologies, Mobility and Security (NTMS)* (2016), pp. 1–5.

[19] Fratantonio, Y., Bianchi, A., Robertson, W., Kirda, E., Kruegel, C., and Vigna, G. TriggerScope: Towards detecting logic bombs in Android applications. In *2016 IEEE Symposium on Security and Privacy (SP)* (2016), pp. 377–396.

[20] Google. Android guides, https://developer.android.com/guide. Accessed August 2022.

[21] Google. Android guide. https://developer.android.com/guide/background/threading, 2023. Accessed January 2024.

[22] Google. Android guide. https://developer.android.com/guide/background, 2023. Accessed January 2024.

[23] Google. Android guide. https://developer.android.com/topic/libraries/architecture/workmanager/migrating-fb, 2023. Accessed January 2024.

[24] Google. Android guide. https://developer.android.com/topic/libraries/architecture/workmanager, 2023. Accessed January 2024.

[25] Google. Android guide. https://developer.android.com/topic/performance/background-optimization, 2023. Accessed January 2024.

[26] Google. Schedule tasks with workmanager. https://developer.android.com/topic/libraries/architecture/workmanager, 2023. Accessed January 2024.

[27] Google. UI/application exerciser monkey. https://developer.android.com/studio/test/other-testing-tools/monkey, 2023. Accessed January 2024.

[28] Google Security Operations. Virus Total free online virus, malware and URL scanner. https://www.virustotal.com, 2022.

[29] Gordon, M. I., Kim, D., Perkins, J. H., Gilham, L., Nguyen, N., and Rinard, M. C. Information flow analysis of Android applications in DroidSafe. In *NDSS* (2015), vol. 15, p. 110.

[30] Guo, C., Zhang, J., Yan, J., Zhang, Z., and Zhang, Y. Characterizing and detecting resource leaks in Android applications. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)* (2013), pp. 389–398.

[31] IDC. Smartphone market share, https://www.idc.com/promo/smartphone-market-share/os, 2024. Accessed January 2024.

[32] Intelligence, S. T., and Team, R. Poseidon's offspring: Charybdis and scylla, https://www.humansecurity.com/learn/blog/poseidons-offspring-charybdis-and-scylla, 2022. Accessed December 2022.

[33] Iqbal, M. App download data, https://www.businessofapps.com/data/app-statistics/, 2023. Accessed January 2024.

[34] Kang, H., Jang, J., Mohaisen, A., and Kim, H. K. Detecting and classifying Android malware using static analysis along with creator information.

International Journal of Distributed Sensor Networks 11, 6 (2015), 479174.

[35] Lhoták, O., and Hendren, L. Scaling Java points-to analysis using SPARK. In *Compiler Construction* (Berlin, Heidelberg, 2003), Springer Berlin Heidelberg, pp. 153–169.

[36] Li, L., Bartel, A., Bissyandé, T. F., Klein, J., Le Traon, Y., Arzt, S., Rasthofer, S., Bodden, E., Octeau, D., and McDaniel, P. Iccta: Detecting inter-component privacy leaks in Android apps. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1* (2015), ICSE '15, p. 280–291.

[37] Li, L., Bissyandé, T. F., Octeau, D., and Klein, J. Droidra: Taming reflection to support whole-program analysis of android apps. In *Proceedings of the 25th International Symposium on Software Testing and Analysis* (2016), ISSTA 2016, p. 318–329.

[38] Li, L., Bissyandé, T. F., Papadakis, M., Rasthofer, S., Bartel, A., Octeau, D., Klein, J., and Traon, Y. L. Static analysis of Android apps: A systematic literature review. *Information and Software Technology 88* (2017), 67 – 95.

[39] Luo, L., Piskachev, G., Krishnamurthy, R., Dolby, J., Bodden, E., and Schäf, M. Model generation for java frameworks. In *2023 IEEE Conference on Software Testing, Verification and Validation (ICST)* (2023), pp. 165–175.

[40] Maji, A. K., Arshad, F. A., Bagchi, S., and Rellermeyer, J. S. An empirical study of the robustness of inter-component communication in Android. In *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2012)* (2012), pp. 1–12.

[41] Octeau, D., Luchaup, D., Dering, M., Jha, S., and McDaniel, P. Composite constant propagation: Application to Android inter-component communication analysis. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1* (2015), ICSE '15, p. 77–88.

[42] Pan, L., Cui, B., Liu, H., Yan, J., Wang, S., Yan, J., and Zhang, J. Static asynchronous component misuse detection for Android applications. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (New York, NY, USA, 2020), ESEC/FSE 2020, Association for Computing Machinery, p. 952–963.

[43] Petsas, T., Voyatzis, G., Athanasopoulos, E., Polychronakis, M., and Ioannidis, S. Rage against the virtual machine: Hindering dynamic analysis of Android malware. In *Proceedings of the Seventh European Workshop on System Security* (2014), EuroSec '14.

[44] Pilgun, A., Gadyatskaya, O., Zhauniarovich, Y., Dashevskyi, S., Kushniarou, A., and Mauw, S. Fine-grained code coverage measurement in automated black-box Android testing. *ACM Trans. Softw. Eng. Methodol. 29*, 4 (July 2020).

[45] Reps, T., Horwitz, S., and Sagiv, M. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (1995), POPL '95, p. 49–61.

[46] Ryder, B. Constructing the call graph of a program. *IEEE Transactions on Software Engineering SE-5*, 3 (1979), 216–226.

[47] Samhi, J. Stack overflow, https://stackoverflow.com/q/70670020/, Jan. 2022. Accessed August 2022.

[48] Samhi, J., Bartel, A., Bissyande, T. F., and Klein, J. RAICC: Revealing atypical inter-component communication in Android apps. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)* (May 2021), pp. 1398–1409.

[49] Samhi, J., Bissyande, T. F., and Klein, J. TriggerZoo: A dataset of Android applications automatically infected with logic bombs. In *2022 IEEE/ACM 19th International Conference on Mining Software Repositories (MSR)* (Los Alamitos, CA, USA, May 2022), IEEE Computer Society, pp. 459–463.

[50] Samhi, J., Gao, J., Daoudi, N., Graux, P., Hoyez, H., Sun, X., Allix, K., Bissyandé, T. F., and Klein, J. JuCify: A step towards Android code unification for enhanced static analysis. In *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)* (May 2022).

[51] Samhi, J., Li, L., Bissyande, T. F., and Klein, J. Difuzer: Uncovering suspicious hidden sensitive operations in Android apps. In *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)* (May 2022), pp. 723–735.

[52] Skylot. JadX: Dex to Java decompiler, https://github.com/skylot/jadx, 2022. Accessed May 2022.

[53] Steensgaard, B. Points-to analysis in almost linear time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (1996), POPL '96, p. 32–41.

[54] Sun, X., Chen, X., Li, L., Cai, H., Grundy, J., Samhi, J., Bissyandé, T. F., and Klein, J. Demystifying hidden sensitive operations in Android apps. In *ACM Transactions on Software Engineering and Methodology* (2022).

[55] Sun, X., Li, L., Bissyandé, T. F., Klein, J., Octeau, D., and Grundy, J. Taming reflection: An essential step toward whole-program analysis of Android apps. *ACM Trans. Softw. Eng. Methodol. 30*, 3 (2021), 36.

[56] Sundaresan, V., Hendren, L., Razafimahefa, C., Vallée-Rai, R., Lam, P., Gagnon, E., and Godin, C. Practical virtual method call resolution for Java. In *Proceedings of the 15th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications* (2000), OOPSLA '00, p. 264–280.

[57] Vallée-Rai, R., Co, P., Gagnon, E., Hendren, L., Lam, P., and Sundaresan, V. Soot: A Java bytecode optimization framework. In *CASCON First Decade High Impact Papers* (USA, 2010), CASCON '10, IBM Corp., p. 214–224.

[58] Vallee-Rai, R., and Hendren, L. J. Jimple: Simplifying Java bytecode for analyses and transformations. Sable Technical Report 1, McGill University, Feb. 1998.

[59] Van Der Veen, V., Bos, H., and Rossow, C. Dynamic analysis of Android malware. *Internet & Web Technology Master thesis, VU University Amsterdam* (2013).

[60] Wei, F., Li, Y., Roy, S., Ou, X., and Zhou, W. Deep ground truth analysis of current Android malware. In *Detection of Intrusions and Malware, and Vulnerability Assessment* (Cham, 2017), Springer International Publishing, pp. 252–276.

[61] Wei, F., Roy, S., Ou, X., and Robby. 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), CCS '14, p. 1329–1341.

[62] Wu, T., Liu, J., Xu, Z., Guo, C., Zhang, Y., Yan, J., and Zhang, J. Light-weight, inter-procedural and callback-aware resource leak detection for

Android apps. *IEEE Transactions on Software Engineering 42*, 11 (2016), 1054–1076.

[63] Xu, L., Zhang, D., Jayasena, N., and Cavazos, J. HADM: Hybrid analysis for detection of malware. In *Proceedings of SAI Intelligent Systems Conference (IntelliSys) 2016* (Cham, 2018), Springer International Publishing, pp. 702–724.

[64] Yan, J., Zhang, S., Liu, Y., Yan, J., and Zhang, J. ICCBot: Fragment-aware and context-sensitive ICC resolution for Android applications. In *2022 IEEE/ACM 44th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)* (2022), pp. 105–109.

[65] Yang, S., Yan, D., Wu, H., Wang, Y., and Rountev, A. Static control-flow analysis of user-driven callbacks in Android applications. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering* (2015), vol. 1, pp. 89–99.

[66] Yang, S., Zhang, H., Wu, H., Wang, Y., Yan, D., and Rountev, A. Static window transition graphs for Android. In *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering* (2015), ASE '15, p. 658–668.

[67] You, W., Liang, B., Li, J., Shi, W., and Zhang, X. Android implicit information flow demystified. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security* (2015), ASIA CCS '15, p. 585–590.

[68] Zheng, C., Zhu, S., Dai, S., Gu, G., Gong, X., Han, X., and Zou, W. 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), SPSM '12, p. 93–104.