Detecting Information Flow by Mutating Input Data

Björn Mathis*, Vitalii Avdiienko*, Ezekiel O. Soremekun*, Marcel Böhme†, and Andreas Zeller*
*Saarland University, Saarbrücken, Germany
{mathis|avdiienko|soremekun|zeller}@st.cs.uni-saarland.de
†National University of Singapore, Singapore
marcel.boehme@nus.edu.sg

***Abstract*—Analyzing information flow is central in assessing the security of applications. However, static and dynamic analyses of information flow are easily challenged by non-available or obscure code. We present a lightweight mutation-based analysis that systematically mutates dynamic values returned by sensitive sources to assess whether the mutation changes the values passed to sensitive sinks. If so, we found a flow between source and sink. In contrast to existing techniques, mutation-based flow analysis does not attempt to identify the specific path of the flow and is thus resilient to obfuscation.**

In its evaluation, our MUTAFLOW prototype for Android programs showed that mutation-based flow analysis is a lightweight yet effective complement to existing tools. Compared to the popular FLOWDROID static analysis tool, MUTAFLOW requires less than 10% of source code lines but has similar accuracy; on 20 tested real-world apps, it is able to detect 75 flows that FLOWDROID misses.

I. INTRODUCTION

When assessing the security of applications, *information flows* play an essential role: Which information sources does the application access, and to which sinks does it send these to? Consequently, *static analysis tools* that detect such information flows see a substantial interest both in practice as in research; for the Android operating system, tools like FLOWDROID [1] or ICCTA [2] represent the state of the art.

Static flow detection tools are effective but they suffer from the principal limitations of static analysis, notably that *all* code must be available for analysis. This problem is illustrated in the example app in Figure 1. The application first accesses sensitive information (A), namely the user's phone number via `getLine1Number()`. This information is then sent via SMS to some third party (B). The flow between A and B can be easily detected by static analysis, and is properly reported by FLOWDROID and ICCTA.

However, obfuscated flows cannot be detected so easily. The example app contains a *native* method, a piece of code that runs directly on the processor and whose source is written in C or C++—in contrast to the Dalvik byte code derived from the Java source. The `devId()` method (D) simply takes a string and returns it. After the sensitive `id` passes through `devId()`, `x` is set and passes into (C); however, FLOWDROID and ICCTA will miss the flow from A to C because it passes through native code, which these tools cannot analyze.

In principle, one could extend static analysis to also consider machine code; and a simple identity function like `devId()` would be easy to recognize. At the machine instruction level,

```
public class HelloJni extends AppCompatActivity {
  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_hello_jni);
    TextView tv =
       (TextView)findViewById(R.id.hello_textview);
    SmsManager sms = SmsManager.getDefault();
    String id = mgr.getLine1Number();      // A
    sms.sendTextMsg("0153", id);           // B
    // ...
    String x = devID(id);
    tv.setText(x);
    // ...
    sms.sendTextMsg("0153", x);            // C
  }

  public native String devID(String id);  // D

  static {
    System.loadLibrary("hello-jni");
  }
}
```

Fig. 1. The HelloJNI Android app uses the Java Native Interface (JNI) to obfuscate a flow. The flow from `getLine1Number()` (A) to `sms.sendTextMsg()` (B) can be detected statically as well as dynamically. However, the flow from A to C can only be found by MUTAFLOW, because `id` flows through the native method `devID()` (D).

though, it is even easier to obfuscate flows, since there is virtually no limit to what the function can do; and any prediction is ultimately thwarted by the halting problem. A static analyser can then either be *optimistic,* and assume nothing bad will happen (as with runtime functions), or be *pessimistic,* and assume anything may happen. Neither resolution is completely satisfactory.

In contrast to static analysis, dynamic analysis allows to analyze concrete executions rather than abstract code. *Dynamic tainting* [3], for instance, tracks data throughout the execution, and would just as well detect a flow from A to B. Finding the flow from A to C via D, though, would require to track data through the hardware or a hardware interpreter, which is no simple feat. Developers wishing to conceal what `devID()` is doing can also resort to *implicit information flow* [4] turning data flow dependencies into control flow dependencies, and again requiring static analysis to identify the alternative control flows. Since existing dynamic and static tools need to analyze the concrete path along which the information travels, they cannot detect deliberately hidden flows, such as from A to C.

In this paper, we investigate a lightweight *mutation-based* alternative to detect information flows. Rather than statically analyzing application code or dynamically tracking data flow, we use an *experimental* approach: We systematically *mutate the information sources of a program to assess whether*

978-1-5386-2684-9/17$15.00 © 2017 IEEE

the mutation impacts its information sinks. Specifically, our MUTAFLOW prototype

1) takes an Android app as well as a set of test cases (given or generated)
2) instruments *sensitive data sources and sinks* in the app to *mutate* input values at sources and track output values at sinks
3) executes tests on unmutated and mutated app versions
4) records the values passed to sensitive sinks, reporting a flow if the value changes due to a mutated sink value.

In our example (Figure 1), MUTAFLOW runs the app twice, the first time unmodified, and the second time injecting a different input value for `getLine1Number()` (A). It then detects that this change causes a change in the calls to `sms.sendTextMsg()` (B) as well as `sms.sendTextMsg()` (C). The previous problem that `id` flows through a native method (D) has no consequences for MUTAFLOW; it only cares about how changes in sources affect sinks, without having to track the path. All changes made by MUTAFLOW simulate changes in the external input data; the actual program functionality is never altered.

The advantage of MUTAFLOW over static analysis approaches is that it hardly overapproximates: It is *sound* in the sense that if it detects a flow, this flow is most likely real. However, as any dynamic approach, it is also *incomplete,* as there may always be executions which exercise flows not previously detected. Hence, MUTAFLOW could be seen as a *complement* to static analysis approaches, focusing on problem areas such as non-analyzable code. However, mutation-based flow analysis can also be seen as an *alternative* analysis, should static analysis not be available or possible. This is because as we show in this paper the *accuracy of* MUTAFLOW *in detecting flows is similar, if not superior, to static analysis tools such as* FLOWDROID *or* ICCTA. This poses mutation-based flow analysis as a new and promising alternative in our portfolio of program analysis techniques.

In summary, this paper makes the following contributions:

1) We introduce *mutation-based flow analysis,* a lightweight program analysis technique to detect information flows.
2) We introduce MUTAFLOW as a prototype implementation for analyzing Android apps (Section II). MUTAFLOW is less than 10% of the size of FLOWDROID.
3) In its evaluation against FLOWDROID (Section III), we find that MUTAFLOW shows comparable performance as FLOWDROID in terms of precision and recall, and is able to detect several flows that FLOWDROID misses (Section IV).

After discussing related work (Section V), we close with conclusion and consequences (Section VI). To facilitate replication, MUTAFLOW and all data from the experiments are available as open source.

II. APPROACH AND IMPLEMENTATION

Mutation-based flow analysis attempts to detect information flow between a *source* a and a *sink* b. Both fundamentals as well as implementation are illustrated using the example in Figure 1.

A. Prerequisites

We start with a program p *and an execution* e*; the execution can either be given (e.g., from a given test case) or generated (e.g., from a test generator).*

MUTAFLOW starts with an *application package* (APK) that contains the app binary as well as all resources to execute it. As tests, MUTAFLOW can use supplied tests as well as leverage the Monkey testing tool [5] to generate executions. A larger number of executions with high coverage of functionality increases the chances of detecting flows.

In our example (Figure 1), the method *onCreate()* is invoked automatically as the app starts—which is actually a plausible attack vector for a malware, in order to collect as much information from as many users as possible.

B. Logging

Given a source a *and a sink* b*, within the execution* e*, we log the concrete values of* a *and* b*, denoted as* a_0 *and* b_0*.*

MUTAFLOW instruments the APK as follows. The instrumenter gets the APK the user wants to analyze and converts it from the compiled code to Jimple code with Soot [6]. Jimple code is a meta-representation of Java code and is used by Soot, a framework with which one can also iterate over the code and inject method calls. In a second step a log-caller and mutation class file we created is compiled with Soot to Jimple code and injected into the decompiled APK. Now we can iterate over the source code line by line and inject methods from this class to write values to the log or mutate source values.

The Soot framework converts the APK into classes, the containing methods and for each method a chain of statements. Now we can iterate over those chains of statements and inject method calls for logging and mutating at each source and sink. These sources and sinks were originally defined by SuSi [7], a tool that detects lines of code where private information flows in or out of the application. We use the pre-computed lists of sources and sinks from SuSi[1].

In our example (Figure 1), the method `TelephonyManager.getLine1Number()` is listed by SuSi as a sensitive source; hence, MUTAFLOW can inject the code

```
Log.write_to_log("Telephony.getLine1Number()");
Log.write_to_log(id);
```

right after the assignment to `id` (A). Likewise, MUTAFLOW can identify the existing logging as sensitive information sinks, and insert the code

```
Log.write_to_log("sms.sendTextMsg()");
Log.write_to_log(id);
```

[1] MUTAFLOW currently handles only Java primitive types and String, thus some sources and sinks in the SuSi set originally used by FLOWDROID are not considered by MUTAFLOW. Specifically, we excluded 73 sources and 63 sinks from the SuSi set which do not return basic types, or are not privacy-invasive, i.e. these sources do not read private information or these sinks cannot be used by malware to send private information from the device. We also added 3 sources that we deem privacy-invasive.

```
public class HelloJni extends AppCompatActivity {
  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_hello_jni);
    TextView tv =
        (TextView)findViewById(R.id.hello_textview);
    String id = mgr.getLine1Number();    // A
    id = Mutator.mutate_string(id);
    Log.write_to_log("Telephony.getLine1Number()");
    Log.write_to_log(id);
    // ...
    Log.write_to_log("sms.sendTextMsg()");
    Log.write_to_log(id);
    sms.sendTextMsg("0153", id);         // B
    // ...
    String x = devId(id);
    tv.setText(x);
    // ...
    Log.write_to_log("sms.sendTextMsg()");
    Log.write_to_log(x);
    sms.sendTextMsg("0153", x);          // C
  }

  public native String devID(String id); // D

  static {
    System.loadLibrary("hello-jni");
  }
}
```

Fig. 2. The HelloJNI program in Figure 1, instrumented by MUTAFLOW.

and

```
Log.write_to_log("sms.sendTextMsg()");
Log.write_to_log(x);
```

before locations B and C, respectively.

C. Mutation

We now generate alternative executions *by mutating the source value. This is done by interposing mutation code into the assignment of a to a_0 such that a_0 is changed to a_0'.*

MUTAFLOW uses instrumentation not only for logging values, but also for mutating source values. To this end, it injects *mutation code* after each information source, which mutates the external input value returned. In most SuSi methods[2], sensitive information is either passed as a string or a number. Hence, MUTAFLOW provides a `Mutator` class that allows to mutate all Java base types including strings (`mutate_strings()`) and numbers (e.g. `mutate_int()`). The string mutator replaces the middle character in the string by another one; the number mutator replaces the numeric value with a random one. However, MUTAFLOW does not prevent violating input pre-conditions. For instance, our `Mutator` class may produce a random source value that violates an input-validation condition, such violation could lead to false positives or reveal exceptional flows to error handling methods.

In our example (Figure 1), MUTAFLOW would inject the mutation code

```
id = Mutator.mutate_string(id);
```

after A and before the (also) injected logging. The fully instrumented program is shown in Figure 2.

[2]SuSi methods are privacy-relevant API calls found by SuSi.

D. Detecting Flows

During the subsequent logging of sinks, we check each sink b for whether its value has changed from b_0 to b_0'. If so, we have shown that there is a flow from a to b.

MUTAFLOW creates multiple versions of the APK, one p with mutation disabled (providing reference values for b_0), and, for each sensitive source a, one p_a with mutation enabled for this source (providing potential values b_0'). It then runs the mutated variants p_a and checks for differences between the b_0 reference values and the b_0' values found in the mutated apps p'. If a value b_0' for some p_a differs from the reference value b_0, then the mutation in a has caused a value change in b; in other words, there was information flow from a to b. MUTAFLOW then reports this flow, including the concrete values (witnesses) a_0, a_0', b_0 and b_0'.

E. Soundness

In the absence of non-determinism, mutation-based flow analysis is conceptually *sound*: It shows that a change in a can cause a change in b, as a precedes b and changing a also changes b—a *counterfactual causality* [8] that also proves the existence of information flow from a to b. This is in contrast to static analysis or dynamic tainting, where most relationships are *possible flows* rather than causal relationships: In `b = zero() * a`, where `zero()` always returns 0, both techniques would detect a possible flow, whereas our approach would fail to find a flow that changes a, a true negative.

Note that in our setting, causality (and thus soundness) requires *perfect reproducibility:* Only if we can ensure that no other input value has changed can we be sure that it was a that caused the change to b. In practice, such perfect reproducibility is hard to achieve due to non-determinism in executions (timing, thread schedules, load, randomness). MUTAFLOW reduces non-determinism by two means. First, MUTAFLOW runs the original app p at least twice. If the values b_0 in the sink b vary across executions, then b is excluded. Second, if MUTAFLOW runs an app variant p_a, the source a is not triggered, but we still observe a difference between b_0' and b_0 for b, then b is excluded.

F. Completeness

Mutation-based flow analysis is *incomplete* in the sense that if it fails to detect a flow, this does not mean the absence of flows. In the code

```
if (hash(input) == 0xdeadbeef)
    output = sensitiveData();
```

for instance, generating an `input` that fulfills the condition is computationally hard; hence, it is unlikely that MUTAFLOW will ever report the flow from `sensitiveData()` to `output`.

An analysis that is *both* sound and complete for real-world apps, reporting all possible flows without false positives, is prevented by the halting problem. In the above example, a static or symbolic analysis can also not know whether the condition can be fulfilled, hence possibly issuing a false alarm.

The halting problem as well as general issues of scale are the reasons why static analysis tools like FLOWDROID or ICCTA cannot claim completeness either.

In practice, ways to address the limitations of incompleteness include:

Have a suite of test cases. In our experiments on real-world apps (Section III-C), we had a student assistant record comprehensive interactions with the apps, which we could replay at will. Such an interaction, which need not take more than 5–10 minutes per app, may already be part of the investigation process[3] . Also, the effort is easily offset by the modest false positive rate in MUTAFLOW; note that the manual investigation of reported flows can easily take 1–2 hours per app.

Integrate with static analysis. Our mutation-based analysis can easily be complemented with static analysis, such that both sets of reported flows are joined, as we do in the evaluation (Section IV). Further integration might guide test generation (and thus MUTAFLOW) towards locations in the code that static analysis has determined to access and propagate sensitive information.

Use run-time sandboxing. If one has a good source of tests, one could easily apply *run-time checks* to disable behavior not seen during testing. During production, the BOXMATE sandboxing approach [10], for instance, would prohibit access to all sensitive sources not accessed during testing (e.g. `sensitiveData()`, above) and thus disable its originating flows. A combination of MUTAFLOW and sandboxing could make MUTAFLOW complete by construction, but may also limit desirable functionality.

G. Implementation Complexity

From the previous description, it should be obvious that MUTAFLOW is a much simpler approach than a full-fledged static analysis for Android, let alone full-fledged symbolic reasoning and checking[4]. To put things into perspective, the full FLOWDROID framework (soot-infoflow-android-develop and soot-infoflow-develop) currently sports ∼36,000 LOC, not counting an additional ∼200,000 LOC for the required Soot, Heros, and Jasmin frameworks.[5] In contrast, MUTAFLOW sports only ∼2,500 lines of Java code, which only is ∼7% of FLOWDROID.

III. EVALUATION DESIGN

To evaluate the effectiveness of MUTAFLOW, we compared our tool with FLOWDROID 1.5, a static taint analysis tool for detecting information flows in Android apps. For our evaluation, we used 131 Android apps in three benchmarks—two *micro-benchmarks* with small apps designed to evaluate information flow detection tools, and one *macro-benchmark* with real-world apps from the Google play store.

Our evaluation addresses three research questions:

RQ1 Effectiveness for micro-benchmarks. Compared to the state-of-the-art, how does MUTAFLOW perform in terms of precision, recall, and F-measure for the micro-benchmarks?

RQ2 Effectiveness for macro-benchmark. Compared to the state-of-the-art, how does MUTAFLOW perform in terms of precision, recall, and F-measure for real world apps from the Google Play market place? Recall that we established the ground truth for real world apps by cross validation rather than by exhaustive means.

RQ3 Performance and Scalability. What is the runtime performance of MUTAFLOW in comparison to the state-of-the-art for both benchmarks?

A. Baseline

FLOWDROID is a highly influential static analysis tool for Android apps, gathering more than 500 citations since its initial release in 2014. Version 1.5 was released in October 2016 and represents the state-of-the-art in information flow detection for Android apps[6]. Like MUTAFLOW, FLOWDROID works directly at the bytecode level and does not require access to the app source code.

B. Micro-Benchmarks

TABLE I
MICRO-BENCHMARKS: DROIDBENCH AND DROIDRA

	Category	#Apps	#Flows	Avg. Size
DroidBench 2.0	Aliasing	1	0	75 LoC
	Android-specific	9	8	45 LoC
	Arrays and Lists	6	2	41 LoC
	Callbacks	14	18	82 LoC
	Emulator Detection	3	6	65 LoC
	Field and Object Sensitivity	7	2	63 LoC
	General Java	20	17	43 LoC
	Implicit Flows	1	1	83 LoC
	Inter-App Communication	2	8	79 LoC
	Inter-Component Communication	14	14	54 LoC
	Lifecycle	16	16	54 LoC
	Reflection	4	4	81 LoC
	Threading	5	5	40 LoC
DroidRA	Reflection	9	9	62 LoC
		111	**110**	**47 LoC**

As **micro-benchmarks**, we chose DroidBench and DroidRA. *DroidBench 2.0* [11] is a collection of 120 small Android apps with several categories of information flows that are obfuscated in one way or another. Version 2.0 of DroidBench significantly extends the micro-benchmark that was originally published with FLOWDROID [1]. *DroidRA* [12] provides more information flows in the reflection categories. Information flows via Java Reflection are particularly hard to discover because functions are not called directly but in a

[3]This process is similar to Apple's manual app review [9]. This review process can be recorded for the application and the recorded review can then be used as input for the MUTAFLOW evaluation.

[4]There are many opportunities to optimize MUTAFLOW. For instance, MUTAFLOW currently monitors the covered sources and sinks for each run, but does not use this information while running the mutated versions. MUTAFLOW also completely rebuilds the application for each mutation.

[5]Java source code only, omitting test code; determined using `cloc $(find -f heros-develop/src -f jasmin-develop/src -f soot-develop/src -f soot-infoflow-android-develop/src -f soot-infoflow-develop | grep '.*.java$')`; retrieved 2017-05-07.

[6]TaintDroid[3] is also a highly influential dynamic analysis tool for Android apps, however it is no longer supported, since it was designed for an outdated Android OS version 4.3, thus it does not work on the Android OS version of our real world apps - Android Marshmallow 6.0.1.

convoluted manner using Java-specific internals, such as class loaders. The average size of an app in the micro-benchmark is 47 Lines of Code (LoC). From DroidBench, we excluded 18 test subjects. Nine apps would crash when executed, due to bugs or missing permissions. For the other nine apps, the respective sources and sinks were excluded because they are not in the SuSi set we used (i.e. they do not use basic types). So, in total we analyze 111 apps in the micro-benchmark. For the remaining test subjects, the categories, and the number of information flows are listed in Table I.

Ground truth. The actual information flows from a specific source to a specific sink are determined manually by investigating the small programs. This establishes the ground truth. As *true positive*, we consider a reported flow that actually exists. As *false positive*, we consider a reported flow that does not actually exist. As *true negative*, we consider an unreported flow that does also not actually exist. As *false negative*, we consider an unreported flow that does actually exist.

Executions. To generate executions for DroidBench, we leverage Google's UI Exerciser Monkey [5]. Monkey takes an Android app and generates a random sequence of user events such as clicks, touches, or gestures, as well as a number of system-level events. In our experiments, we fix the length of a sequence at 10,000 user events and run one test sequence for each variant of an app. Since Monkey is essentially a random test generation tool, *we repeat each experiment 10 times* with *different* random seeds to gain statistical power. However, within one experiment, we use the *same* test sequence for all (mutated) variants of the app. For DroidBench, Monkey demonstrates that MUTAFLOW does not depend on pre-existing test cases. However, for more complicated usage scenario (e.g. log-in scenario) sophisticated input-generators would be required to reveal flows.

TABLE II
MACRO-BENCHMARK: APPS FROM THE GOOGLE PLAY STORE

Name	Jimple LoC	Size in Bytes
Mysugr	590 kLoC	12.8 MB
Ab Workouts	572 kLoC	6.3 MB
Adidas miCoach	599 kLoC	41.7 MB
Fitness at Home	400 kLoC	18.8 MB
7 Minute Workout	515 kLoC	6.9 MB
Fast Calorie Counter	340 kLoC	2.4 MB
Water Drink Reminder	595 kLoC	9.9 MB
Abs Workout 7 Minutes	344 kLoC	4.7 MB
BMI and Weight Tracker	324 kLoC	4.1 MB
Fabulous – Motivate Me!	715 kLoC	20.3 MB
Test Diabetes Sugar-Joke	367 kLoC	6.9 MB
Kegel Trainer – Exercises	448 kLoC	6.8 MB
Fitness Recipe of the Day	351 kLoC	2.0 MB
Lifesum: Healthy lifestyle	678 kLoC	31.5 MB
Calorie, Carb & Fat Counter	409 kLoC	10.9 MB
30 Day Butt Challenge FREE	569 kLoC	4.6 MB
Blood Pressure Log (bpresso)	282 kLoC	3.2 MB
30 Day Fit Challenges Workout	119 kLoC	7.1 MB
Calorie Counter—FDDB Extender	674 kLoC	7.3 MB
Runkeeper—GPS Track Run Walk	618 kLoC	39.3 MB
Sum	9,509 kLoC	247.5 MB

C. Macro-Benchmark

As **macro-benchmark**, we chose 20 random apps from the Google Play store—by first randomly selecting a category: "Health & Fitness" and then randomly selecting 20 apps from this category. Table II provide more details about these real world apps; in the absence of source code, "LoC" refers to the length of the decompiled Jimple code.

1) Decompile app with `JADX`
2) Open the code in Android Studio (so we can use features like finding the usage of methods)
3) Find the source and the sink reported from the log (the logs provide information about containing class and method)
4) For each source:
   a) If the value flows into the return of a method, the method usages have to be checked
   b) If the value flows into a call parameter, the called method has to be checked
   c) If the value flows into a field, the read usages of the field have to be checked
5) For each sink:
   a) If the value comes from the methods parameter, the usage of the method has to be checked
   b) If the value comes from a field, the write usages of the field have to be checked
6) A flow is found if a feasible path between source and sink is found (e.g. there must not be any checks that prevent the path to be taken)
7) For MUTAFLOW, the log can also be consulted:
   a) If a value occurs only once in the mutated execution but the API method is still called in all execution, the flow is also categorized as true positive
   b) If the mutated value is found in plain text in the sink, the flow is categorized as true positive

Fig. 3. Policy for manual classification

Ground truth. Unlike for the micro-benchmark, for the real-world apps we cannot obtain the absolute ground truth but we can cross-validate. If there are only 100 sources and 100 sinks, we would need to manually check $20 \times 100 \times 100 = 200{,}000$ potential flows to identify the complete set of true information flows for all apps. This is clearly impractical. However, on a best effort basis we *manually checked all reported information flows* in order to distinguish true from false positives. To mitigate experimenter bias, we follow a strict *coding protocol* involving the *independent classification* by two researchers *R1* and *R2*:[7]

1) *R1* and *R2* agree on a policy how to classify the reported flows into true and false positives.
2) *R1* classifies all flows and refines the coding policy.
3) *R1* and *R2* discuss the refined policy.
4) *R2* (*independently*) classifies all flows.
5) *R1* and *R2* check the rate of agreement.
6) If the rate is too low, they discuss policy and recode.
7) Otherwise, they proceed to resolve any contentions.

The final policy used is listed in Figure 3.

To retrieve the source code from the Android apps, we used the Dex to Java decompiler `JADX` [17]. We consider as false *negatives* all flows that are true positives for one technique but not reported by the other. For instance, if FLOWDROID reported an information flow from source A to sink B, we first manually checked whether there really does exist an

[7]Coding is a methodology from grounded theory that is used in sociology and psychology to evaluate qualitative properties [13]. In the context of software engineering [14], [15], this methodology is also referred to as *open card sort* [16].

information flow from A to B and then checked whether MUTAFLOW also reports the same information flow. If it did not, the flow was marked as false negative for MUTAFLOW. The manual validation was done by inspecting the source code of each app. If the path of a reported flow was not feasible, for instance due to constants, dynamic types, or other influences, then the flow was categorized as false positive.

Executions. Almost all health apps require sign-up, log-in or otherwise entering specific data, which cannot be synthesized by the Monkey test generator. We therefore had a *user generate test sequences*. Specifically, we hired a student assistant whose task was to click through the app with the intention to explore most of its features, while his interaction would be recorded. The student would sign-up and log-in as needed and enter the data that was required to proceed to the next user dialog. The recorded test sequences can be replayed deterministically at will. For each app, there is one sequence with a length of up to 30 user interactions.

D. Physical Setup and Infrastructure

Being a static analysis, FLOWDROID can execute on an arbitrary machine that has access to the app byte code; in contrast, MUTAFLOW executes the app on an Android device, emulated or real:

- We execute FLOWDROID on one of our compute servers with 144 cores and 700 GB of RAM.[8]
- For MUTAFLOW running the micro-benchmarks, we use the *Android emulator* on a PC,[9] emulating a Nexus 5 running Android Marshmallow 6.0. We only ran one single emulator on the machine, as we found parallel emulators interfering with each other.
- For MUTAFLOW running the macro-benchmarks, we use *a single Android device* with six cores and 2GB of RAM,[10] controlled by a server via the Android Debug Bridge (ADB). We use a real device instead of an emulator for two reasons. First, real world apps often cannot be installed on an emulator, as it lacks features such as the Google play store; second, real devices report realistic values for all sources and sinks.

As a 144-core compute server is way more powerful than an Android device, let alone an emulated one, the increase in computing power might seem generous towards FLOWDROID; however, it corresponds to a realistic setting where a user might have a lot of computing power but access to only one Android device during the execution of MUTAFLOW.

IV. EVALUATION RESULTS

A. Effectiveness for Micro-Benchmarks

We start with RQ1: **How effective are MUTAFLOW and FLOWDROID on our set of micro-benchmarks?**

[8]Specifically, a 4 × Intel(R) Xeon(R) CPU E7-8867 v4 @ 2.40GHz with 144 virtual cores (Intel Hyperthreading), running Debian 3.16 Linux.

[9]Specifically, an Intel i7 4770S with 8 virtual cores with 32 GB RAM running Ubuntu 14.04 LTS.

[10]Specifically, a a Nexus 5X that has a 64-bit Adreno 418 GPU and a Qualcomm Snapdragon 808 Processor @ 1.8GHz with 6 cores and 2GB of main memory, running Android Marshmallow 6.0.1.

1) Accuracy: Table III shows the results for FLOWDROID, whereas Table IV shows the results for MUTAFLOW. (Note that the MUTAFLOW results are averaged over 10 runs.)

TABLE III
ACCURACY OF FLOWDROID ON MICRO-BENCHMARKS

	Classified as			
Input	**Flow**	**No Flow**	**Total**	
Flow	TP = 64	FN = 46	110	*Precision* = 86%
No Flow	FP = 10	TN = 12	22	*Recall* = 58%
Total	74	58	132	*Accuracy* = 58%
				F-Measure = 70%

TABLE IV
ACCURACY OF MUTAFLOW ON MICRO-BENCHMARKS

	Classified as			
Input	**Flow**	**No Flow**	**Total**	
Flow	TP = 74.3	FN = 35.7	110	*Precision* = 98%
No Flow	FP = 1.5	TN = 19.8	21.3	*Recall* = 68%
Total	75.8	55.5	131.3	*Accuracy* = 72%
				F-Measure = 80%

We see that on average, MUTAFLOW reports only 1.5 false positives[11], whereas FLOWDROID reports 10 false positives (6 times as many) .

> With a precision of 98%, *almost all flows reported by* MUTAFLOW *are actual flows.*

Interestingly, the *recall* of MUTAFLOW is higher, too; MUTAFLOW detects 68% of all flows, whereas FLOWDROID detects 58%. The higher accuracy of MUTAFLOW over FLOWDROID is also indicated by the measures of accuracy and F-measure.

> MUTAFLOW *exhibits a better precision, recall, and accuracy than the state-of-the-art,* FLOWDROID.

So, why is the precision of MUTAFLOW "only" 98% if, in principle, it should be 100%? The reason again is the non-determinism, as discussed in Section II-E. Some tests are *flaky* in the sense that executing the same test case twice might give different results. This flakyness stems from the randomness that is inherent to the Android environment. For instance, when a time stamp is appended to a message, it might seem as if the monitored sink that receives the message is impacted by a mutated source while it is not.

2) Complementarity: The aim of MUTAFLOW is not to be an alternative to FLOWDROID, but rather *complement* it—and this makes perfect sense, as each technique can find flows the other does not. As shown in Figure 4, without MUTAFLOW, FLOWDROID would find only 58% of the existing flows. Using both techniques, 90% of all existing flows would be detected. Averaged over ten runs, MUTAFLOW finds 35.5 actual flows that FLOWDROID does not find, while FLOWDROID reports 25.2 flows that MUTAFLOW does not find.

3) Strengths of FLOWDROID *over* MUTAFLOW*:* What are the strengths and weaknesses of each technique? Figure 6 summarizes the detection rates for the individual categories. We see that MUTAFLOW *detects fewer flows than* FLOWDROID in the categories

[11]For 5 applications, MUTAFLOW had at least one false positive.

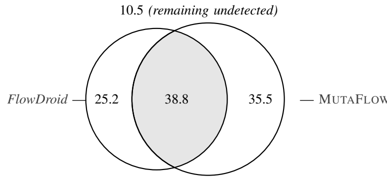

Fig. 4. Venn Diagram showing the intersection of found flows (true positives) for FlowDroid and MUTAFLOW on the micro-benchmark.

```
1 @Override
2 protected void onCreate(Bundle savedInstanceState) {
3     super.onCreate(savedInstanceState); // [...]
4 
5     // Acquire a reference
6     // to the system Location Manager
7     locationManager = (LocationManager)
8         getSystemService(Context.LOCATION_SERVICE);
9 
10    // Register the listener with the Location
11    // Manager to receive location updates
12    locationManager.requestLocationUpdates(
13        LocationManager.GPS_PROVIDER, 5000, 10,
14        locationListener);
15 }
```

Fig. 5. The *AnonymousClass1* benchmark uses callbacks to send out location changes. FLOWDROID detects the associated flow, but MUTAFLOW misses it because the emulator takes too long to report the changed location.

- callbacks,
- emulator detection, and
- inter-app communication.

We explain the reduced effectiveness by the fact that MUTAFLOW requires test cases that actually trigger the flow. For instance, to detect the flows in the *callbacks*-category, Monkey would need to click a specific sequence of buttons in a specific order. The probability that our random testing tool makes the right sequence of clicks decreases exponentially with the length of the required sequence. To detect the *inter-application flows*, Monkey would need to open one app, trigger the source, close the app, open the other app, and trigger the sink. With Monkey as automated test generation tool, MUTAFLOW can detect only one of six inter-application flows.

Again, we illustrate the strength of one tool over the other using an example. In the DroidBench app *AnonymousClass1*, Figure 5 shows the essential function, registering a callback handler for changed locations. In the MUTAFLOW setting, the emulator does change the location of the device during testing, but the emulator takes too long to report the change to the app; hence, the callback is never called, and MUTAFLOW cannot detect the dynamic flow.

4) Strengths of MUTAFLOW *over* FLOWDROID*:* Let us now go back to Figure 6. We see that MUTAFLOW *detects more flows* than FLOWDROID for

- implicit flows,
- inter-component communication, and
- the reflection category.

We explain the improved performance with FLOWDROID's difficulty to analyze indirect flows along convoluted paths. Unlike FLOWDROID, MUTAFLOW is ignorant of the specific path along which an important information travels. If there is a test case that exercises both the source and the sink, then it is quite likely that MUTAFLOW detects the flow. Hence,

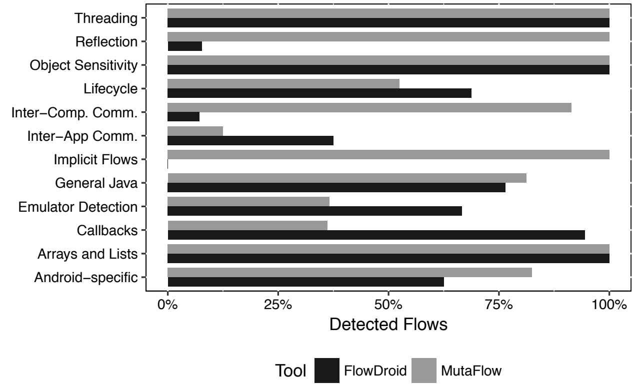

Fig. 6. Histogram showing the number of detected flows as percentage of the total number of flows per category for both, MUTAFLOW and FlowDroid.

```
16 private String obfuscateIMEI(String imei){
17     String result = "";
18 
19     for(char c : imei.toCharArray()){
20         switch(c){
21             case '0' : result += 'a'; break;
22             case '1' : result += 'b'; break;
23             case '2' : result += 'c'; break;
24             case '3' : result += 'd'; break;
25             case '4' : result += 'e'; break;
26             case '5' : result += 'f'; break;
27             case '6' : result += 'g'; break;
28             case '7' : result += 'h'; break;
29             case '8' : result += 'i'; break;
30             case '9' : result += 'j'; break;
31             default: // [...]
32         }
33     }
34     return result;
35 }
```

Fig. 7. The *ImplicitFlow1* benchmark obfuscates a sensitive device identifier. The implicit flow is missed by FLOWDROID, but detected by MUTAFLOW.

MUTAFLOW performs well for *implicit flows*, i.e., where the data is modified or obfuscated along the path, for *inter-component communication*, i.e., where intents or activities are used to communicate between different components of the same app, and for *reflection*, where Java-specific calls to the reflection framework are used to construct and send messages.

As a typical example of a data flow ignored by FLOWDROID, but detected by MUTAFLOW, consider the DroidBench app *ImplicitFlow1*. Figure 7 shows the essential function, obfuscating a sensitive device identifier. Since there is no explicit flow, i.e., a direct assignment of any data in `imei` to `result`, FLOWDROID misses the flow. (Note that *dynamic tainting* approaches such as TaintDroid [3] would also miss such implicit flows for the same reason.) MUTAFLOW, however, easily detects the flow since any change to `imei` also results in a change in `result`; this change then propagates to the sensitive sink, where MUTAFLOW can detect it.

B. Effectiveness for Real World Apps

Let us now turn to real-world applications and address RQ2: **How effective are MUTAFLOW and FLOWDROID on our set of macro-benchmarks?** In the remainder of this section we discuss our results listed in Table V.

TABLE V
ANALYSIS RESULTS ON THE MACRO-BENCHMARK

	Analysis time (seconds)			Number of flows							
	FLOWDROID	MUTAFLOW		FLOWDROID				MUTAFLOW			
Name	**analysis**	**instr**	**exec**	**sum**	**TP**	**FP**	**unknown**	**sum**	**TP**	**FP**	**unknown**
Mysugr	crash ↯	355	13,724	0	0	0	0	23	23	0	0
Ab Workouts	90	519	9,739	9	1	6	2	3.5	2	1.5	0
Adidas miCoach	1,628	259	6,608	11	9	1	1	4	4	0	0
Fitness at Home	43	265	3,479	8	2	0	6	0	0	0	0
7 Minute Workout	crash ↯	328	4,368	0	0	0	0	2.25	1	1.25	0
Fast Calorie Counter	19	135	2,106	0	0	0	0	0	0	0	0
Water Drink Reminder	timeout ↯	643	16,095	0	0	0	0	13.25	13.25	0	0
Abs Workout 7 Minutes	crash ↯	165	3,280	0	0	0	0	1	1	0	0
BMI and Weight Tracker	out of mem ↯	141	2,202	0	0	0	0	0	0	0	0
Fabulous – Motivate Me!	timeout ↯	475	6,124	0	0	0	0	1	1	0	0
Test Diabetes Sugar-Joke	40	194	1,698	1	1	0	0	0	0	0	0
Kegel Trainer – Exercises	21	214	2,293	0	0	0	0	0	0	0	0
Fitness Recipe of the Day	13	153	837	0	0	0	0	0	0	0	0
Lifesum: Healthy lifestyle	crash ↯	867	15,422	0	0	0	0	0	0	0	0
Calorie, Carb & Fat Counter	32	286	6,856	2	2	0	0	4.5	4.5	0	0
30 Day Butt Challenge FREE	286	393	4,138	60	0	0	60	0	0	0	0
Blood Pressure Log (bpresso)	16	208	3,695	11	4	7	0	2	2	0	0
30 Day Fit Challenges Workout	26	49	845	0	0	0	0	0	0	0	0
Calorie Counter—FDDB Extender	crash ↯	585	11,961	0	0	0	0	0	0	0	0
Runkeeper—GPS Track Run Walk [12]	crash ↯	1623	timeout ↯	0	0	0	0	27	25	2	0
Sum	2,217	7,866	149,856	102	19	14	69	81.5	76.75	4.75	0
Average (w/o ↯ in MUTAFLOW)	202	329	6,077								
Average (w/o ↯ in FLOWDROID)	202	243	3,845								

Running FLOWDROID and MUTAFLOW on the macro-benchmark not only took considerable time; we also encountered a large number of crashes and timeouts.[13] When FLOWDROID crashes, it does not report any flows; hence, the respective set of flows found is empty.

Following our process for establishing ground truth manually Section III-C, it took us between one and two hours per app and person to validate the reported flows by FLOWDROID or MUTAFLOW; for MUTAFLOW, validation was easier as we had an execution with concrete values to examine.

For a significant number of flows reported by FLOWDROID, we could not determine whether they were true positives or false positives, due to their complexity. Sixty uncategorizable flows (87%) where reported by FLOWDROID for the *30 Day Butt Challenge FREE* app. These flows went through a very large hashmap that is used throughout the app. If a single tainted value flows into a hashmap, FLOWDROID marks the complete hashmap as tainted, spreading the taints throughout the program. We believe that most flows are false positives but conservatively mark them as uncategorizable.

The *Runkeeper* app is special in that it drove tools and humans to their limits. FLOWDROID crashed on it and MUTAFLOW was not done after 10 hours of testing. For MUTAFLOW, we would make use of the flows found until the timeout. In Runkeeper, MUTAFLOW detected 12 flows that originated from a sensitive source, ended in a SQL database, and later impact a sensitive sink; here, we assumed that the flows went through the database. All numbers are reported in Table V.

1) Accuracy: Table VI summarizes the results for FLOWDROID, whereas Table VII summarizes the results for MUTAFLOW. (Note that MUTAFLOW results are averaged over four runs, in order to account for the inherent non-determinism in the Android environment.)

TABLE VI
ACCURACY OF FLOWDROID ON MACRO-BENCHMARK

	Classified as			
Input	**Flow**	**No Flow**	**Total**	
Flow	TP = 19	FN = 74.75	93.75	*Precision* = 58%
No Flow	FP = 14	TN = 4.75	18.75	*Recall* = 20%
Total	33	79.5	112.5	*Accuracy* = 21%
				F-Measure = 30%

69 flows could not get categorized (60 flows arise from 1 app)

TABLE VII
ACCURACY OF MUTAFLOW ON MACRO-BENCHMARK

	Classified as			
Input	**Flow**	**No Flow**	**Total**	
Flow	TP = 76.75	FN = 17	93.75	*Precision* = 94%
No Flow	FP = 4.75	TN = 14	18.75	*Recall* = 82%
Total	81.5	31	112.5	*Accuracy* = 81%
				F-Measure = 88%

The results are in line with those already seen for the micro-benchmark (Section IV-A): Most notably, MUTAFLOW sports a precision of 94%, whereas with FLOWDROID, only 58% of flows reported are true positives. Given the effort it takes to manually identify a flow as true or false positive, a high precision is definitely an important goal.

> *On our set of real-world apps, 94% of all flows reported by* MUTAFLOW *are actual flows.*

Considering the total set S of true positives (reported by either tool and manually classified as actual flow), MUTAFLOW

[12] Unlike the other apps in our macro-benchmark, Runkeeper was executed only once, due to changes in the back-end login authentication of the app. After our first execution, we discovered that our human-generated test cases for this version of Runkeeper could no longer be executed, because we could no longer login into this version of the app.

[13] All bugs encountered in FLOWDROID have been reported.

reports 82% of these, whereas FLOWDROID reports only 20%; here, the low recall of FLOWDROID is easily explained by completing the analysis only for 11 out of the 20 apps. However, one can see that the high precision of MUTAFLOW is not offset by a low recall, as indicated by the high overall F-measure. In our macro-benchmark, MUTAFLOW does not report any flow for 10 apps, because of the following reasons: the use of a small set of SuSi sources and sinks, inability to trigger certain sources (e.g. due to lack of sophisticated test cases), and non-determinism at certain sinks.

2) Complementarity: As already indicated by the false negative numbers, again each tool misses flows that would be reported by the other one. Figure 8 shows the found flows for each of the tools; and again, we see how both approaches complement each other in their respective strengths.

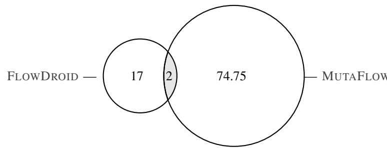

Fig. 8. Venn Diagram showing the intersection of found flows (true positives) for FlowDroid and MUTAFLOW on the macro-benchmark.

C. Performance and Scalability

Let us now close the evaluation with RQ3: **What is the performance of MUTAFLOW, and how does it compare to FLOWDROID?** As already discussed in Section III-D, the machines we use for MUTAFLOW (a single Android device, real or emulated) and FLOWDROID (a 144-core compute server) are very different, so we may well compare peanuts and pumpkins here. Still, as compute servers are still way more common than large farms of Android devices, such a setting may well represent the typically available distribution of computing power.

1) Performance on the Micro-Benchmark: On our *micro-benchmark,* FLOWDROID takes an average time of 4.9s per application. In contrast, MUTAFLOW takes 9.9s per application. The longer time of MUTAFLOW is attributed to the overhead it takes to instrument, install, and execute an application, as well as the performance penalty of the emulator; FLOWDROID need only analyze the (very short) byte code of each app.

> *On our micro-benchmark, both* FLOWDROID *and* MUTAFLOW *are very fast, with an average time of 4.9s and 9.9s per app, respectively.*

2) Performance on the Macro-Benchmark: On our *macro-benchmark,* performance is a more interesting story. Table V lists the time taken by FLOWDROID (first column) vs. the time taken by MUTAFLOW, whose time is split into instrumentation (second column) and actual test execution (third column). Looking at the times, let us only consider the 11 apps where FLOWDROID could determine the flows. For these apps, FLOWDROID is very fast, with an average running time of 202 seconds, or 3.5 minutes; MUTAFLOW is about 20× slower, taking on average 243 seconds (4 minutes) for creating the mutated app versions; and ∼1 hour (3,845 seconds) per app for running the tests on the individual mutants. Over all 20 apps (including the 10 hour timeout for Runkeeper), MUTAFLOW takes an average of 7,493 seconds, or ∼2 hours.

> *On our set of real-world apps, the* MUTAFLOW *analysis takes 1–2 hours of analysis per app and device.*

However, keep in mind that FLOWDROID is running on a 144-core compute server, whereas MUTAFLOW runs on a single Android device. Both mutant creation and test execution are embarrassingly parallel problems, and easily distributed across multiple devices. A rack of 20 Android devices would reduce the average MUTAFLOW testing time down to 3 minutes, and thus easily catch up with FLOWDROID—and still be a much smaller investment than a compute server. Again, for the practical analysis of information flows, we would recommend to have both compute servers for static analysis as well as testing devices for checking concrete flows and mutations.

> *Mutation-based flow analysis is embarrassingly parallel.*

D. Threats to Validity

Like any empirical investigation, our evaluation is subject to threats to validity. The first concern is *external validity,* and notably *generality.* First, the efficiency of mutation-based flow analysis and static analysis, respectively, is dependent on a large set of factors, including analyzability of the code, the effort it takes to identify sources and sinks, the ability to automatically test the code, the effort it takes to create a test, the time it takes to run a test, the value of true positives, and the cost of false positives. Hence, our results do not generalize to arbitrary programs, and the choice of which method(s) to use will always be left to the user. The aim of this work is simply to point out mutation-based flow analysis as a relatively simple alternative that enriches the portfolio of program analysis.

Regarding *internal validity,* our investigation of the flows may be subject to *researcher bias,* that is, we may consciously or unconsciously favor the results of our own tool over the FLOWDROID alternative. For the *macro-benchmark*, we counter this threat by following a strict coding policy, as detailed in Section III-C; for the *micro-benchmark,* this threat is countered by having the benchmark as well as its ground truth all being constructed by the FLOWDROID team, who, if at all, would have a bias towards demonstrating the power of their tool. Both tools are provided with the same set of sensitive sources and sinks. All our data and assessments are available for replication and scrutiny (Section VI).

V. RELATED WORK

Work related to mutation-based flow analysis falls into three categories.

Static analysis. Static information flow analysis attempts to detect (sensitive) information flows from static code

analysis. FLOWDROID [1] is the most influential representative in the Android area, and the gold standard for detecting flows; notable extensions include ICCTA [2] to analyze inter-process communication and DroidRA [12] to handle reflection. The recent DFlow [18] system is a flow detection alternative that focuses on scalability and precision. In contrast to all these static code analyses, MUTAFLOW is a *dynamic* experimental approach, and thus can detect flows that static code analysis cannot, as discussed in Section II-E (Soundness), and vice versa (Section II-F; Completeness). This is also the subject of our evaluation (Section III and Section IV).

Dynamic analysis. Dynamic information flow analysis tracks data as it is being processed through an execution. The most influential representative in the Android area is TaintDroid [3]. Its dynamic tainting tags sensitive input data with a label ("taint"), which is passed along to further variables in each computation that involves a tainted variable; if tainted data reaches a sensitive sink, the tool reports a flow. As dynamic flow analysis can considerably slow down program execution, researchers also have searched for correlations between inputs and outputs [19]. In contrast to these approaches, MUTAFLOW is an *experimental* approach which shows true counterfactual causality and thus soundness by construction (Section II-E).

Experimental analysis. Experimental program analysis techniques [20] introduce a change in the program execution and determine its impact. MUTAFLOW, as its name suggests, is inspired by *mutation analysis,* where artificial defects are introduced into the code to determine whether they will be caught by a test; it is most related to the JAVALANCHE approach [21], which determines the impact of the change in the remaining execution. Given a specific statement, SENSA [22] modifies the statements during test execution in order to determine and quanitify the impact of this statement on the original execution. The ORBS approach [23] selectively removes program statements to determine a reduced program that observationally behaves the same as the original program w.r.t. to a slicing criterion. However, all of these techniques substantially change the original execution by altering the program rather than the input coming from an information source, which is arguably minimally invasive. Moreover, none of these approaches is geared towards detecting the existence of flows at analysis time, as mutation-based flow analysis is.

VI. CONCLUSION

To detect information flows, it can already suffice to mutate an input from a sensitive source and to see whether, while keeping everything else unchanged, this change impacts some value passed to some sensitive sink. Mutation-based flow analysis may seem annoyingly simple, but it is very effective: It can reveal flows that static analysis cannot detect; and where a static analysis tool is not available, not possible, or crashes, it may even serve as a simple alternative. Mutation-based flow analysis thus complements and augments state-of-the-art analysis tools.

In contrast to static analysis, mutation-based flow analysis requires an execution and thus input data—either generated or manually crafted. This requirement is offset by having to spend little to no effort on false positives: By construction, mutation-based flow analysis achieves near-perfect precision, meaning that close to 100% of reported flows are actual ones. In the long run, we see static analysis and mutation-based analysis tools work hand in hand, such that they further strengthen their respective findings.

Besides general improvements such as performance or stability, our future work will focus on the interplay between static analysis and mutation-based analysis:

Focused test generation. Rather than using a pure random testing tool such as Monkey, one could guide a directed test generator [24] towards code where static analysis already has determined the existence of potential flows. Static analysis could also tell a test generator which values to provide for which source, and which code to execute in order to reach a particular sink.

Mutations at function level. The impact of mutations at sensitive sources can also be tracked at other locations in the program code, not only sensitive sinks. In Figure 1, this can help to establish the information flow within the `devId()` function; in Figure 7, this shows that there is information flow through the `obfuscateIMEI()` function. Such information at the function level not only gives more detail about the actual information flow, it also provides important *function summaries* for static analysis: If FLOWDROID knows from MUTAFLOW that `obfuscateIMEI()` has information flow from input to its return value, FLOWDROID need no longer miss the overall flow.

Intertwined analyses. The future of program analysis lies in the integration of several techniques: Static analysis, dynamic analysis, test generation, experimental approaches as well as symbolic approaches all must work hand in hand to mitigate their respective weaknesses, and turn their integration into strength. Only with a broad knowledge and an open mind can we defeat today's and tomorrow's challenges of program analysis.

MUTAFLOW and all experimental data referred to in this paper are available as open source; see our package at

https://github.com/anosubmission/mutaflow-data

ACKNOWLEDGMENT

Our utmost thanks go to Siegfried Rasthofer, Steven Arzt, Eric Bodden, and the entire FLOWDROID team for making all of their tool chain and benchmark data available and keeping it up to date for application, replication, and assessment. Their support has been exemplary in every aspect.

REFERENCES

[1] 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*, ser. PLDI '14, 2014, pp. 259–269.

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

[3] 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 real-time privacy monitoring on smartphones," *Communications of the ACM*, vol. 57, no. 3, pp. 99–106, Mar. 2014.

[4] W. You, B. Liang, J. Li, W. Shi, and X. Zhang, "Android implicit information flow demystified," in *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*, ser. ASIA CCS '15. New York, NY, USA: ACM, 2015, pp. 585–590. [Online]. Available: http://doi.acm.org/10.1145/2714576.2714604

[5] Website, "UI/Application Exerciser Monkey," https://developer.android.com/studio/test/monkey.html, online; accessed 25-April-2017.

[6] P. Lam, E. Bodden, O. Lhoták, and L. Hendren, "The Soot framework for Java program analysis: a retrospective," in *Cetus Users and Compiler Infastructure Workshop (CETUS 2011)*, vol. 15, 2011, p. 35.

[7] S. Arzt, S. Rasthofer, and E. Bodden, "SuSi: A Tool for the Fully Automated Classification and Categorization of Android Sources and Sinks," Technical Report. SuSi source code available at https://github.com/secure-software-engineering/SuSi, 2013.

[8] D. Lewis, "Causation," *Journal of Philosophy*, vol. 70, no. 17, pp. 556–567, 1973.

[9] Website, "Macworld," https://www.macworld.com/article/2047567/how-apple-is-improving-mobile-app-security.html, online; accessed 25-April-2017.

[10] K. Jamrozik, P. von Styp-Rekowsky, and A. Zeller, "Mining sandboxes," in *Proceedings of the 38th International Conference on Software Engineering*, ser. ICSE '16. New York, NY, USA: ACM, 2016, pp. 37–48. [Online]. Available: http://doi.acm.org/10.1145/2884781.2884782

[11] Website, "Droidbench 2.0," https://github.com/secure-software-engineering/DroidBench, online; accessed 25-April-2017.

[12] 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*, ser. ISSTA 2016. New York, NY, USA: ACM, 2016, pp. 318–329. [Online]. Available: http://doi.acm.org/10.1145/2931037.2931044

[13] K. Charmaz, *Constructing grounded theory: a practical guide through qualitative analysis*. London; Thousand Oaks, Calif.: Sage Publications, 2006.

[14] D. Lo, N. Nagappan, and T. Zimmermann, "How practitioners perceive the relevance of software engineering research," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2015, 2015, pp. 415–425.

[15] A. Begel and T. Zimmermann, "Analyze this! 145 questions for data scientists in software engineering," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014, 2014, pp. 12–23.

[16] W. Hudson, *Card Sorting In "The Encyclopedia of Human-Computer Interaction, 2nd Ed."*. Interaction Design Foundation, 2013.

[17] Website, "JADX: Dex to Java decompiler," https://github.com/skylot/jadx, online; accessed 25-April-2017.

[18] W. Huang, Y. Dong, A. Milanova, and J. Dolby, "Scalable and precise taint analysis for android," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ser. ISSTA 2015. New York, NY, USA: ACM, 2015, pp. 106–117. [Online]. Available: http://doi.acm.org/10.1145/2771783.2771803

[19] J. Huang, X. Zhang, and L. Tan, "Detecting sensitive data disclosure via bi-directional text correlation analysis," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2016. New York, NY, USA: ACM, 2016, pp. 169–180. [Online]. Available: http://doi.acm.org/10.1145/2950290.2950348

[20] J. R. Ruthruff, S. Elbaum, and G. Rothermel, "Experimental program analysis: A new program analysis paradigm," in *Proceedings of the 2006 International Symposium on Software Testing and Analysis*, ser. ISSTA '06. New York, NY, USA: ACM, 2006, pp. 49–60. [Online]. Available: http://doi.acm.org/10.1145/1146238.1146245

[21] D. Schuler and A. Zeller, "Javalanche: Efficient mutation testing for Java," in *Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ser. ESEC/FSE '09. New York, NY, USA: ACM, 2009, pp. 297–298. [Online]. Available: http://doi.acm.org/10.1145/1595696.1595750

[22] H. Cai, S. Jiang, R. Santelices, Y. J. Zhang, and Y. Zhang, "Sensa: Sensitivity analysis for quantitative change-impact prediction," in *2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation*, ser. SCAM '14, Sept 2014, pp. 165–174.

[23] D. Binkley, N. Gold, M. Harman, S. Islam, J. Krinke, and S. Yoo, "Orbs: Language-independent program slicing," in *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2014, 2014, pp. 109–120.

[24] M. Böhme, V.-T. Pham, M.-D. Nguyen, and A. Roychoudhury, "Directed greybox fuzzing," in *Proceedings of the 24th ACM Conference on Computer and Communications Security*, ser. CCS, 2017, pp. 1–16.