

Technical Report

Nr. TUD-CS-2015-0031

Feb. 5th, 2015

Harvesting Runtime Data in Android Applications for Identifying Malware and Enhancing Code Analysis



TECHNISCHE
UNIVERSITÄT
DARMSTADT



EC SPRIDE
EUROPEAN CENTER FOR
SECURITY AND PRIVACY BY DESIGN

Authors

Siegfried Rasthofer

Steven Arzt

Marc Miltenberger

Eric Bodden

EC SPRIDE / Fraunhofer SIT & Technische Universität Darmstadt

Harvesting Runtime Data in Android Applications for Identifying Malware and Enhancing Code Analysis

Siegfried Rasthofer, Steven Arzt, Marc Miltenberger, and Eric Bodden
Secure Software Engineering Group
European Center for Security and Privacy by Design (EC SPRIDE)
Technische Universität Darmstadt & Fraunhofer SIT
Darmstadt, Germany

Abstract—It is generally a challenging task to tell apart malware from benign applications: obfuscation and string encryption, used by malware as well as goodware, often render static analyses ineffective. In addition, malware frequently tricks dynamic analyses by detecting the execution environment emulated by the analysis tool and then refraining from malicious behavior.

In this work, however, we present HARVESTER, a novel approach that combines a variation of program slicing with dynamic execution, and show that it can be highly effective in the triage of current mobile malware families. For this malware, HARVESTER allows a fully automatic extraction of runtime values from any position in the Android bytecode. Target phone numbers and messages of SMS messages, decryption keys or concrete URLs that are called inside an Android application can usually be extracted even if the application is highly obfuscated, and even if the application uses anti-analysis techniques (e.g., emulator detection or delayed execution / “time bombs”), dynamic code loading and native method calls for string decryption. As we show, HARVESTER not only aids human malware analysts, but also acts as an automatic deobfuscation tool that reverts the introduction of encrypted strings and reflective method calls as they are often introduced by obfuscators such as DexGuard.

We will make available HARVESTER as open source. Experiments on 13,502 current malware samples show that HARVESTER can extract many sensitive values from applications, usually in under one minute, and this fully automatically and without requiring the simulation of UI actions. Our results further show that HARVESTER’s deobfuscation can enhance existing static and dynamic analyses, for instance with FlowDroid and TaintDroid.

I. INTRODUCTION

Android has long become the most popular smartphone operating system with about one billion installations worldwide and 1.5 million new devices activated every day [3]. According to a study conducted by Gartner Inc., this gives Android a total market share of more than 80% [23]. One reason for the success of the platform is the availability of applications for almost every need: More than one million applications are now available from various vendors. While this abundance of applications is very convenient for the user, it also makes it much harder to decide whether or not an app should be installed, since the trustworthiness and competence of the developer is usually hard to judge, as is the quality of the application itself. Even the official Google Play market is not devoid of malicious applications or apps containing grave security flaws [11], [28].

To assess the quality or security of an Android application, experts are frequently interested in its runtime values. Knowing runtime values is important in many aspects. For instances, the analyst often needs to know which URLs data is transmitted to, which phone numbers SMS messages are sent to, what the contents of these messages are, and which databases the app reads from the phone (contacts, e-mail, SMS messages, etc.). If an application, for instance, opens *http* instead of *https* connections, this can hint at a security flaw. If SMS messages are sent to well-known premium-rate scamming numbers, this indicates malware. However, even if applications are not outright malicious, many of them are obfuscated to protect their respective author’s intellectual property, nevertheless hindering manual inspection. The app’s bytecode often contains string constants only as encrypted byte sequences. The analyst then has to manually investigate this particular app decrypts those strings at runtime and reconstruct every such string—a very costly and labor-intensive undertaking that must be repeated for every new application [27].

Obfuscation also poses challenges for automated static-analysis tools. In modern Android malware such as Obad [42] or FakeInstaller [37], methods are often not called directly but through reflection, with the target method’s name being stored in an encrypted format. The actual call-target string is computed not before runtime, which makes it unavailable to classic static-analysis tools. Without this information, however, these tools have an incomplete picture of the code’s behavior, effectively hindering malware detection. Inter-component or inter-application analyses such as EPICC [31] face similar challenges. In Android, inter-application and inter-component communication is usually performed using so-called *intents*, where the target can be specified as a string. If this string is obfuscated, static analyses can no longer determine the intent’s recipient. The availability of powerful obfuscation tools such as DexGuard [41] or ProGuard [18], the latter of which is even shipped with the Android SDK, aggravate the problem as they help spreading such obfuscation techniques.

At runtime, all required values are eventually computed, and thus become detectable. Nevertheless, dynamic analyses aiming at extracting those values also face challenges when analyzing real-world malware samples. Some analyses, for instance, rely on intercepting malicious network traffic through man-in-the-middle proxies [5]. This only works, however, if the code paths that produce the respective network traffic are actually executed

at runtime. But generally neither the tool nor the analyst can cover all possible execution paths in a finite amount of time. Furthermore, the code may behave differently depending on the execution environment (emulator vs. real phone, different phone models, etc.). In fact, malicious applications nowadays use so-called time bombs, which cause the malware to be executed only at a certain time, or date or logic bombs which activate the malware based on some environmental trigger [10], [33], [45]. This also includes bot-net malware that only acts in response to a command received from a command-and-control server—a command that dynamic analysis tools will miss. Equally important, Android applications are heavily interactive. To trigger the malicious behavior certain user interactions may be required. Dynamic tools need to simulate these interactions, as they can gather information only about code paths that they actually execute. Previous work [9], [21], [36] has shown that because of this even for medium-sized Android apps complete code coverage is often impossible to achieve. Consequently, many runtime values of interest remain unknown when using purely dynamic tools. This makes it very difficult for automatic classifiers or human analysts to detect malicious behavior.

In this work, though, we show that by combining a particular variation of traditional static-analysis algorithms known from program slicing with dynamic code execution one can build an approach that allows for the fully automatic extraction of most interesting runtime values for current Android malware applications. This includes even sophisticated malware families such as Obad, Pincer, or FakeInstaller. Our tool HARVESTER works on Android bytecode, requiring no source code of the analyzed app. The approach is hybrid. Using slicing, it first statically isolates all program code that contributes to the computation of a value of interest; all other bytecode instructions that do not influence this value are discarded. Crucially, though, our static slicing algorithm differs in some respect from the traditional algorithm as proposed by Weiser and others [4], [32], [46]. Their definitions are sound based on a fixed environment for code execution. HARVESTER deviates from this soundness concept by generating parametric slices that emulate the behavior of the original program in different environments, in particular on actual devices opposed to emulators. Using traditional slicing would require a precise model each of these environments which is impractical for real-world Android systems, for instance because every smartphone has a different number, different accounts, different contacts, etc. HARVESTER thus instead simulates not the environment values, but the reactions to different environments the target app implements. If an app, for instance, changes its behavior based on a command from a command-and-control server in a botnet, HARVESTER creates a parametric slice that allows each of these behaviors to be triggered explicitly without having to model the botnet communication as such.

In a second step, HARVESTER executes the statically extracted program slices dynamically, on an unmodified Android emulator or stock Android phone. It explicitly triggers all the different behaviors of the parametric slice which allows the complete reconstruction of the values of interest, decrypting any encrypted values along the way. HARVESTER directly instruments the reporting mechanism for the values of interest into the slices, so no changes to the runtime environment (emulator, Android firmware, etc.) are necessary. Moreover, HARVESTER invokes each slice directly, regardless of its

original position in the application code. By the way the slices are constructed, at least for all current malware we analyzed, this eliminates the need for UI interaction or other external input during slice execution. In these situations, HARVESTER thus overcomes the problem of limited code coverage that exist with classical UI-testing approaches. Security analysts can also use the slices computed by HARVESTER to improve existing analyses such as TaintDroid [13]. Since TaintDroid can only detect a data flow if the respective code is actually executed, directly invoking a slice avoids the same code-coverage problem of classical testing tools.

In a third step, HARVESTER integrates the dynamically discovered values directly into the application. This means that code statements that use reflection, for instance, are complemented by statements that directly call the target method originally called through the reflection API. As we show, this injection allows off-the-shelf static analyses such as FlowDroid [17] to correctly interpret formerly obfuscated behaviors that they would otherwise not be able to detect—without any modification to the analysis tool.

Given enough knowledge, skill, time and resources, malware authors can eventually break all known deobfuscation approaches, including HARVESTER's. The goal we wanted to achieve with HARVESTER, though, was to derive an analysis and deobfuscation approach that is effective at least for all *currently* popular malware families. We thus conducted an evaluation with 13,502 current malware samples. For these samples, HARVESTER not only discovers runtime value in 99% of all cases of interest but also proves efficient enough for mass analyses of Android applications. On average, on a single malware application HARVESTER takes less than one minute to extract concrete telephone numbers and text messages of a potential SMS trojan application. During the course of the experiments, HARVESTER reported many interesting runtime values, such as command-and-control messages and addresses, and successfully deobfuscated malware which hides sensitive API calls through reflections. Furthermore, HARVESTER successfully extracts the obfuscated key used by the well-known WhatsApp messaging app [34] to encrypt its message store.

In summary, this paper presents a novel hybrid information-extraction approach for Android applications based on the following original contributions:

- a variation of traditional slicing algorithms fine-tuned to optimally support the hybrid extraction of runtime values in currently known malware,
- a dynamic execution system for running the computed code slices and extracting the values of interest without user interaction,
- an augmentation algorithm for re-injecting the obtained data values into the application as constants to enable further analysis with other off-the-shelf tools or by human analysts, and
- an evaluation of the approach's feasibility for a mass-analysis on current real-world malware applications.

The remainder of this paper is structured as follows: Section II motivates our work with the example of obfuscated Android malware. In Section III, we explain the architecture of

our approach and the algorithms used to compute the runtime values. The implementation of the HARVESTER tool is discussed in Section IV, whereas Section V shows the results of our experimental evaluation. Section VI gives an overview of related work and Section VII concludes the paper.

II. MOTIVATING EXAMPLE

Listing 1 shows a real-world code snippet taken from FakeInstaller [37]¹, one of the most widespread malware families according to a recent list of top 10 Android malware applications [15]. Note that we decompiled the sample to Java source code and that we added code comments to ease the understanding. FakeInstaller heavily relies on obfuscation to hide its behavior from both analysis tools and manual investigators. The obfuscator generates random class and method names, eliminating most semantic information. Furthermore, at runtime, instead of calling methods directly, FakeInstaller takes a string previously encrypted and decrypts it using a lookup table. It then uses reflection to find the class and method that bear the decrypted name and to finally invoke the retrieved method.

For instance, the string constant `VRIf3+In9a.aTA3RYnD1BcVRV]af` in line 6 is decrypted to the name of the `android.telephony.SmsManager` operating-system class which is then loaded using reflection. `SmsManager` is a sensitive resource: malware can use it to send expensive premium-rate SMS messages at the cost of the user. Indeed, the string `BaRIta*9caBBV]a` in line 9 is decrypted to `sendMessage`, the name of the method for sending text messages, which is also located using reflection. Line 15 finally invokes the method, actually sending out the text message. With the help of these runtime values we have just discovered that the method `gdadbjrj` is responsible for sending text messages where `paramString1` is the number and `paramString2` is the body of the message.

Many current malware applications are obfuscated in a similar way, either manually or by using commercial tools such as DexGuard [41]. A human analyst would have to

carefully inspect the decompiled bytecode, find the lookup table, and manually decrypt all strings to detect the behavior described above. Strings decrypted once cannot usually be reused, as different malware variants use different lookup tables. Many static-analysis tools would not be able to find the call to `sendMessage` at all because they do not interpret string operations nor the reflection API, and thus lack the corresponding call-graph edge. And even those static-analysis tools that do model these APIs can be fooled by attackers who encode the strings using native-code libraries and the static analysis fails to model.

But the code in the example challenges dynamic-analysis approaches just as well. First, they have to find an execution path actually triggering the `gdadbjrj` method. If, for instance, method `gdadbjrj` is only executed after a delay (time bomb) or after a specific environment trigger (logic bomb) [10], this is not a trivial undertaking. In such situations, the analysis never knows when it is safe to stop the dynamic test execution and cannot easily speed up analysis either. Other malware might call the malicious code only when the user clicks on a certain button. Then, the analysis tool must be able to emulate this button click and all user actions required to reach the user-interface state displaying the button in the first place. Various obfuscation techniques for dynamic approaches, such as emulator-detection mechanisms [33], [35], [45] complicate this analysis even further. The check in line 3, for instance, prevents the malicious code from being executed if the execution environment shows characteristics of an emulator such as the presence of certain files or a specific timing behavior. It also aborts if a debugger is attached to the application. Dynamic analysis environments can never fully hide all of these characteristics [10] and thus fail on sophisticated malware.

HARVESTER, on the other hand, fully automatically retrieves all relevant runtime values of the example in Listing 1. The security analyst simply specifies the variables for which runtime values should be retrieved. In a first step, knowing nothing else about the app, in this example the analyst would likely choose the first parameters to the `forName` (line 11) and `getMethod` calls and the two variables `paramString1` and `paramString2` (line 14) to learn more about the reflective calls. HARVESTER's static slicer then automatically extracts all code computing those values, while *crucially*, however, discarding certain conditional control-flow constructs that do not impact the computed value. (We give details later.) In the example, this will discard the emulator-detection check at line 3, and also outside of `gdadbjrj` will only retain exactly the code that computes the input values `paramString1` and `paramString2`. HARVESTER's dynamic component then runs only the reduced code. Since all emulator-detection checks are eliminated, the dynamic analysis immediately executes all those parts of `gdadbjrj` relevant to the computation of the selected values. At runtime, the analysis discovers the name `SmsManager.sendMessage` of the method called through reflection. Lastly, it reports the concrete telephone numbers (7151, 2858 and 9151) and bodies (701369431072588745752, 7012394196732588741192 and 7834194455582588771822) of the SMS messages sent.

Note that HARVESTER does not require any manipulations to the underlying Android framework. It works purely on the bytecode level of the target application, through a bytecode-to-

¹Sample MD5: dd40531493f53456c3b22ed0bf3e20ef

```

1 public static boolean gdadbjrj(String paramString1,
2   String paramString2){ [...]
3   // Emulator check: Evade dynamic analysis
4   if (zhfdghfdgd()) return;
5   // Get class instance
6   Class clz = Class.forName(gdadbjrj.gdadbjrj
7     ("VRIf3+In9a.aTA3RYnD1BcVRV]af"));
8   Object localObject = clz.getMethod(
9     gdadbjrj.gdadbjrj("]a9maFVM.9"), new
10    Class[0]).invoke(null, new Object[0]);
11  // Get method name
12  String s = gdadbjrj.gdadbjrj("BaRIta*9caBBV]a");
13  // Build parameter list
14  Class c = Class.forName(gdadbjrj.gdadbjrj
15    ("VRIf3+InVTnSaRI+R]KR9aR9"));
16  Class[] arr = new Class[] {
17    nglpsq.cbhgc, nglpsq.cbhgc, nglpsq.cbhgc, c, c };
18  // Get method and invoke it
19  clz.getMethod(s, arr).invoke(localObject, new
20    Object[] { paramString1, null, paramString2, null,
21    null });
22 }

```

Listing 1: Highly obfuscated code sending a text message (FakeInstaller [37] malware family)

bytecode transformation. To aid further static analysis of the app’s code, HARVESTER also manifests the concrete targets of reflective calls into the original applications, as direct method calls. As we show in Section III-C, this enables further analysis using existing off-the-shelf static-analysis tools such as static taint analyses. The fact that HARVESTER removes unnecessary runtime checks (e.g., emulator detection) from the original application allows security analysts to apply dynamic-analysis tools such as TaintDroid, which the original application would have recognized and fooled, to the converted application.

III.SOLUTION ARCHITECTURE

Figure 1 depicts HARVESTER’s general architecture. The general usage scenario for HARVESTER is to compute so-called *values of interest* for which the following two definitions are required.

Definition 1: A logging point $\langle v, n \rangle$ is a pair consisting of a variable or field access v and an arbitrary statement s given that v is in scope at s .

Definition 2: A *value of interest* is the concrete value of variable v at a logging point $\langle v, n \rangle$.

For instance, if one is interested in runtime values passed to a conditional check

```
s: if (a.equals(b))
```

the runtime values of a and b are both *values of interest* at this statement s , inducing the two logging points $\langle a, s \rangle$ and $\langle b, s \rangle$. Another example would be an API call to the `sendMessage` method such as

```
s: sendMessage(arg1, arg2, arg3, arg4, arg5)
```

where $\langle arg1, s \rangle$ and $\langle arg3, s \rangle$ are the logging points at the method-call s . The corresponding runtime values are the *values of interest*.

To compute such values of interest, HARVESTER first reads the APK file and a configuration file defining the logging points. Theoretically, the same approach could also be applied to Java class files, but here we present and evaluate it for Android APK files.

The first part of the analysis is a static backwards-slicing computation starting at these code points, as will be further explained in Section III-A. This slicing step runs on a desktop computer or computation server. The pre-computed slices are then used to construct a new, reduced APK file which contains only the code required to compute the values of interest, and an executor activity. The task of the executor activity is to invoke the computed slices and report the computed values of interest. This new, reduced APK file is then executed on a stock Android emulator or real phone, as we explain in Section III-B. These steps are fully automated and no user interaction is required.

Optionally, the security analyst can use off-the-shelf dynamic-analysis tools such as TaintDroid to screen the reduced APK further. This is beneficial over analyzing the original APK, as the reduced APK will only execute the behavior of interest. For instance, if an analyst is interested in potential data leakages, he can use HARVESTER to produce a new apk, which directly executes the slice that potentially leaks some sensitive

information. Alternatively, the analyst can instruct Harvester to inject the discovered runtime values into the original application (Section III-C). This enables existing off-the-shelf static-analysis tools such as CHEX [25], SCanDroid [2] or FlowDroid [17] to further analyze the application. This is advantageous over analyzing the original application as runtime values and reflective method calls are now statically embedded as constants respectively normal method calls in the application, allowing the tools to discover those calls and values.

In some highly-obfuscated applications, the logging point cannot directly be identified. Assume that the application uses reflection to call the method that sends out a text message and the analyst is interested in the telephone number to which the text message gets sent. In this case, he would need to run HARVESTER twice. In the first round, he retrieves the targets of all reflective method calls. With this information, he can then identify the actual logging point for the second round.

The remainder of this section will explain the static backwards slicing and the runtime execution phase in more detail.

A. Static Backward Slicing

Figure 2 shows a simplified control flow graph of an example program. When the user clicks on the “ClickMe” button, method 1 is called which in turn invokes method 3 at some point. Method 3 contains the logging point, i.e., the position at which a value of interest shall be retrieved. The analyst wants to compute this value of interest irrespective of its original position in the program and without having to emulate the button click. Even if method 3, for instance, is only invoked under certain circumstances (time or logic bombs), the value shall nevertheless be retrieved directly. To achieve this, HARVESTER uses *program slicing* to extract the code which computes the values of interest (nodes printed in solid black in Figure 2) so that this code can then be run in isolation irrespective of its original position in the control flow.

In traditional slicing as defined by Weiser [46], a program

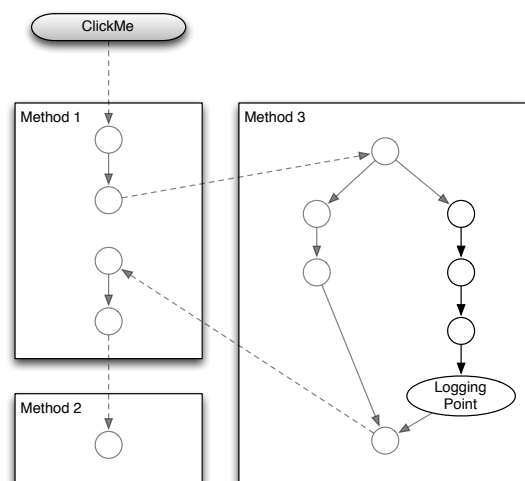


Fig. 2: Important Statements in Backwards Slice

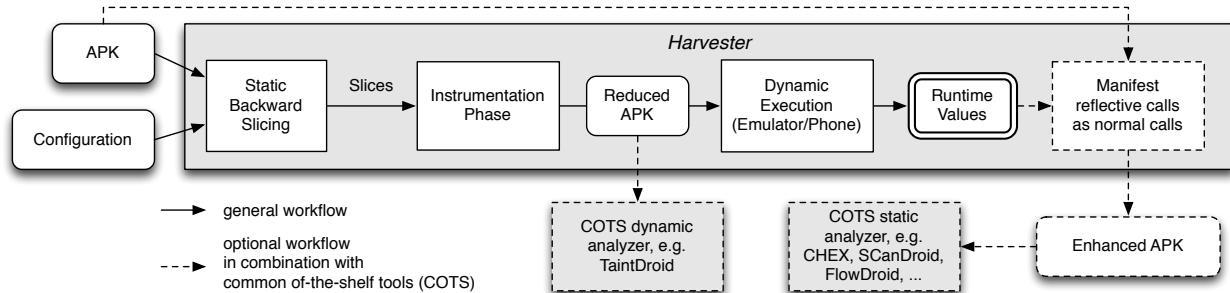


Fig. 1: Workflow of the HARVESTER Approach

```

1 String number = null;
2 String logInfo = null;
3
4 String deviceId = getDeviceId();
5 if(deviceId.equals("0000000000000000")) {
6     return;
7 }
8
9 String model = Build.MODEL;
10 if(!model.equals("generic")) {
11     if(simCountryIso().equals("RU")) {
12         number = "5623";
13         logInfo = "country iso: RU";
14     }
15     if(simCountryIso().equals("US")) {
16         number = "7823";
17         logInfo = "country iso: US";
18     }
19     if(simCountryIso().equals("DE")) {
20         number = "9371";
21         logInfo = "country iso: DE";
22     }
23 }
24
25 if(number != null)
26     sendTextMessage(number, "msg");

```

Listing 2: Original Code: Emulator-Checks and Premium-SMS

```

1 String number = null;
2 /*
3
4
5     removed by slicer
6
7 */
8
9 String model = Build.MODEL;
10 if(!model.equals("generic")) {
11     if(simCountryIso().equals("RU")) {
12         number = "5623";
13         // removed by slicer
14     }
15     if(simCountryIso().equals("US")) {
16         number = "7823";
17         // removed by slicer
18     }
19     if(simCountryIso().equals("DE")) {
20         number = "9371";
21         // removed by slicer
22     }
23 }
24
25 // removed by slicer
26 sendTextMessage(number, "msg");

```

Listing 3: Traditional Slicing for *number* at Line 26

```

1 String number = null;
2 /*
3
4
5     removed by slicer
6
7 */
8
9 String model = Build.MODEL;
10 if(EXECUTOR_1) {
11     if(EXECUTOR_2) {
12         number = "5623";
13         // removed by slicer
14     }
15     if(EXECUTOR_3) {
16         number = "7823";
17         // removed by slicer
18     }
19     if(EXECUTOR_4) {
20         number = "9371";
21         // removed by slicer
22     }
23 }
24
25 // removed by slicer
26 sendTextMessage(number, "msg");

```

Listing 4: Parametric Slicing for *number* at Line 26

slice S is an *executable program* that is obtained from a program P by removing statements, such that S replicates the behavior of P [43] with respect to the *slicing criterion*. In our example, methods 1 and 2 are irrelevant and can safely be removed since all computations required for it are local to method 3. Extracting the respective code from method 3 and directly invoking it during the later execution phase (see Section III-B) not only significantly reduces the size of the code to be executed and thus makes the execution faster, but also removes the need to click on the “ClickMe” button. This works because computing a specific value of interest at a given position typically only requires the execution of a small fraction of the original program that is in close proximity to the logging point.

In the following, we will show why this traditional style of slicing is insufficient for our task at hand. Since real-world obfuscated malware code such as the one shown in Section II is too complicated to demonstrate the slicing, we introduce the artificial example presented in Listing 2. In the example, the malware sends a text message to a premium rate number in line 26. The target telephone number is stored in variable *number*. Depending on the country of the mobile carrier, different phone numbers are used (Lines 11, 15 and 19). The

contents of the variable *logInfo* are not relevant for sending the premium-rate message.

The example has been inspired by the Pincer malware family which scans for various system properties and refrains from the malicious behavior if traces of an emulator are found. Such checks form a very common pattern which is used in other malware families as well. In the example, such checks are carried out in two different places: Line 5 and 10.

Listing 3 shows the output of a traditional slicer for the example program. All references to the unrelated *logInfo* variable have been removed. The first emulator check in Line 5 has also been removed together with all computations that were only necessary to carry out this single check. This happens because the conditional in Line 5 only defines *if* the target telephone number is computed, not *how*. The slicing criterion has no data dependency on this check. This reasoning applies to most time bombs and logic bombs which can thus safely be removed without affecting the outcome of the computation of the value of interest.

Note, though, that when executing the slice in Listing 3 within an emulator, variable *number* will remain *null*, since

the emulator check in line 10 will fail. Traditional slicing aims at constructing a slice that mimics the behavior of the original program *in the current environment*. In this current (emulated) environment it is correct, to compute `null` for the number, but to the security analyst, who wants to know all possible target phone numbers, it is not helpful. More generally, the analyst wishes to retrieve all values of interest that can be computed in any environment, regardless of the (typically emulated) environment in which the slice is currently being executed. With a traditional slice, the analyst would still have to emulate all these possible environments. As environment dependencies in general, and emulator checks in particular, can exist in a wide variety of ways [33], [45], simulating all possible combinations of these environment variables is infeasible in practice.

To cope with this problem, HARVESTER extends program slicing with a special treatment of conditionals. For those conditionals that influence *which* value is computed, at runtime HARVESTER must explore both branches as each branch potentially defines a different runtime value for the respective variable of interest. To force a specific path to execute, HARVESTER replaces all conditionals contained in the slice with static Boolean expressions as shown in Listing 4. This makes the slice *parametric* with at least one instance of every possible value of interest.² When later executing the slice (see Section III-B), HARVESTER can then run the example once for each of the 16 possible combinations of the Boolean values `EXECUTOR_1` through `EXECUTOR_4`. This enumerates all possible outcomes of the computation for `number`.

Applying this principle to real-world apps, however, leads to a number of challenges which we present using different code examples in the remainder of this section.

²In general, multiple instances of the parametric slice can lead to the same value of interest.

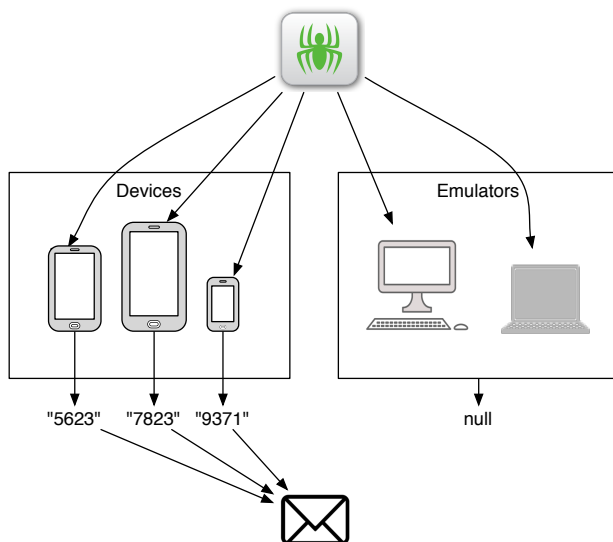


Fig. 3: Retrieving Values of Interest from Different Environments

a) Internal dependencies within a slice: When replacing conditionals, it is important not to break the semantics of the target program. Ideally, for every value of interest computed by HARVESTER, there should exist a possible runtime environment in which the original program would have computed the same value.

Listing 5 shows a more complex computation for the value of interest at logging point $(x, 12)$. If one would blindly replace all conditionals in this example, the slice shown in Listing 6 would compute the value 1 in some case (`EXECUTOR_0` set to `true`) and not terminate in all other ones (`EXECUTOR_0` set to `false`). The correct values computed by the original program (16, 11, 42) would never be computed by the parametric slice. This happens because the Boolean values remain constant for each execution, i.e., an instance of the parametric slice will either always take the left branch or always take the right branch for the respective conditional.

In program verification, such problems are often addressed by *unrolling loops* [16]. Since HARVESTER, however, prepares a parameterized slice for dynamic execution, unrolling the loops up to a sufficiently large number of iterations is generally infeasible. If the number is too low, this causes HARVESTER to miss the actual output of the computation. If it is too high, this leads to a combinatorial explosion of the possible Boolean values and resulting paths: for n conditionals, up to 2^n paths can be explored.

HARVESTER therefore follows a different approach, not unrolling loops. Instead, conditionals that only have data dependencies on the values of interest, i.e., are internal dependencies in the slice, are not replaced with Boolean variables in the first place. They are retained as they are in the original program. In Listing 5, this means that the conditionals in Lines 5 and 6 are replaced with `EXECUTOR_1` and `EXECUTOR_2` respectively, while expressions in line 3 and 8, which depend on x , are kept untouched. The result is shown in Listing 7. This way, HARVESTER can explicitly steer the execution into specific paths while keeping intact the computation of the values of interest in most cases. The following paragraph shows a case in which the replacement of conditionals can nevertheless lead to a wrong value.

b) Missing value of interest: In some cases, replacing the original conditionals with Boolean variables may cause the slice to compute a different value than the original program. More precisely, there may be environments in which the original program computed a value that cannot be computed by any instance of the parametric slice. In Listing 8, the logging point $(phoneNo, 4)$ has no direct data-flow dependency on the loop counter i . Therefore, HARVESTER replaces the condition in Line 2 with a static Boolean expression `EXECUTOR_0` which forces the execution once into skipping the loop completely and once into looping infinitely, disregarding the original semantics. In neither case will the execution compute the original value 123.

It is at this point where HARVESTER has to cut corners and has to restrict itself to the typical cases observed in the wild. To avoid termination problems, HARVESTER limits the maximum iteration count to a fixed value and aborts the loop if this limit is exceeded. Though this may lead to false results, it does not seem to pose problems for analyzing state-of-the-art

```

1 int x = 1;
2 while (true) {
3     if (x >= 10)
4         break;
5     if (a) x = x * 2;
6     else if (b) {
7         x = x + 3;
8         if (x % 2 == 0) x = x * 2;
9     }
10    else x = 42;
11 }
12 send(x);

```

Listing 5: Branching on internal value

```

1 int x = 1;
2 while (true) {
3     if (EXECUTOR_0)
4         break;
5     if (EXECUTOR_1) x = x * 2;
6     else if (EXECUTOR_2) {
7         x = x + 3;
8         if (EXECUTOR_3) x = x * 2;
9     }
10    else x = 42;
11 }
12 send(x);

```

Listing 6: Simple transformation with internal branching

```

1 int x = 1;
2 while (true) {
3     if (x >= 10)
4         break;
5     if (EXECUTOR_1) x = x * 2;
6     else if (EXECUTOR_2) {
7         x = x + 3;
8         if (x % 2 == 0) x = x * 2;
9     }
10    else x = 42;
11 }
12 send(x);

```

Listing 7: Internal branching issue resolved

malware samples as shown in Section V. Most obfuscation and decryption algorithms used by current malware manipulate values from the code or from a file. They do not use loops to dynamically generate completely new values like in the example.

c) Spurious value of interest: Since HARVESTER over-approximates the paths to be executed, it may return false positives, i.e., values that could not be computed by the original program in any given environment. In Listing 9, the code computes a different telephone number for every mobile carrier country. The code assigning the value 0000, however, can never be reached in the original program because there is no environment with an XX country code. Since HARVESTER cannot make any such assumptions about the possible set of environments, it explores this path as well and returns the spurious value. In practice it seems that the amount of such spurious information will be low and not put any significant burden on the security analyst.

d) Android API Handling: Most applications use API classes such as `SharedPreferences` to store data, which is later retrieved and then e.g., sent out to the internet. Storage and retrieval can be distributed among the program, such as being executed when different buttons in the user interface are clicked. A slicing approach that does not model this data dependency between user actions would yield an incorrect slice that tries to read non-existent data from an uninitialized data store. To handle these cases, HARVESTER resolves all calls that write to persistent storage and prepends them to the slice. This approximation may, however, miss some of the data if the stored value is ambiguous, as only the last value is

```

1 int phoneNo = 120;
2 for (int i = 0; i < 3; i++)
3     phoneNo++;
4 send("" + phoneNo, "Hello");

```

Listing 8: Lost Value of Interest

```

1 String number = null;
2
3 if (simCountryIso().equals("DE")) {
4     number = 9371;
5 }
6
7 if (simCountryIso().equals("XX")) {
8     number = 0000;
9 }
10
11 sendTextMessage(number, "msg");

```

Listing 9: Path Over-Approximation

retained and all earlier values are overwritten. While handling the external storage better is an interesting area for future work, our experiments confirm that our current solution still produces values for all logging points.

Further special handling is required for API calls that access environment values such as free-text user input. Since the slice will automatically be executed without user interaction in phase 2 (see Section III-B), HARVESTER injects dummy values instead of the actual API calls that read out the UI. This prevents the slices from crashing even if they still access some external resources. Our evaluation has shown that this is not a problem in practice (see Section V).

Note that HARVESTER can also cope with dynamic code loading and native methods, as long as all code containing the logging points is contained within the APK’s bytecode at instrumentation time. If, for instance, the value of an SMS message is computed by calling a dynamically loaded function using reflection, or by invoking a native method, the slicer will declare this function as required and the dynamic pass will execute the function as any other function, providing the same implementation that would also be invoked during normal app execution.

B. Dynamically Executing the Reduced APK

Every slice computed during the static slicing phase yields a new method in the reduced APK file produced by HARVESTER. The executor activity injected into the same APK file calls all these methods one after another, directly after the new app has been started on an unmodified emulator or a stock Android phone. The executor writes the computed runtime values into an SQLite database on the device’s SD card that can then be downloaded and evaluated on a desktop computer. Since the slices are executed directly, regardless of their original position in the application code, HARVESTER requires no user interaction that might otherwise be necessary to reach the code location of the computing statements. If, for instance, the extracted code was originally contained in a button-click handler, it would have required the user or an automated test driver to click that button to be executed. HARVESTER, however, executes the sliced code directly, making this unnecessary. In fact, the reduced app does not even contain any GUI elements from the original app. The reduced app is packaged with the same resources as the original app, such that code that would load encrypted strings, for instance, from external resources, will find those resources also in the reduced APK.

As explained in Section III-A, slices are parametric and

HARVESTER must explore every possible combination of branches to retrieve the values of interest that can be computed in all possible environments. For the executor, this means that it must set all possible combinations of these Boolean values and re-run the code slice. In general, this leads to 2^n paths where n is the number of conditionals between the introduction of the variable and the position of the logging point. In practice, however, n is very limited. In the few cases in which it is not, many of those paths will yield the same value. HARVESTER therefore supports randomly picking a predefined maximum number of slice instances (i.e., combinations of the Boolean variables) to execute. While this may generally lead to missed values of interest, in our experiments this point showed not to be a limitation.

C. Injecting Runtime Values in the original APK

Existing static-analysis approaches usually rely on a call graph to determine which target method a method invocation actually causes to execute. For the large fraction of malware applications that are obfuscated using reflective method calls, such as the example in Listing 1, call-graph construction fails. Some tools do not support reflective calls at all during call-graph construction, while others like Soot [44] do support resolving reflective calls with constant target strings, but cannot resolve dynamically constructed method or class names. HARVESTER, however, can aid those off-the-shelf tools by manifesting the runtime values of reflective call targets resolved during the dynamic execution as ordinary method calls in the application’s bytecode. This allows existing call-graph construction algorithms to construct a sound call graph with ease.

Method `obfuscated()` in Listing 10 shows a simplified example of a reflective method call. Assume that HARVESTER detects two different possible runtime values for variable `t`, namely `"foo"` and `"bar"`. HARVESTER replaces method `obfuscated` with method `directCalls` that contains direct call edges to these two target methods. To allow for cases in which there are further call targets which HARVESTER failed to detect dynamically, the old reflective call is nevertheless retained in the fall-through branch (line 10).³ Off-the-shelf analysis tools such as CHEX [25], SCanDroid [2] or FlowDroid [17] can then analyze the enriched APK file without requiring special handling for reflection or string operations used to build the target method name. To the best of our knowledge, HARVESTER is the first fully-automated approach that performs such a value injection on the Android platform. The enriched APK files are functionally equivalent to their respective originals and only use normal application-level code. Running them does not require any changes to the operating system or the emulator.

Our prototype currently supports this technique for reflective method calls only, but it could easily be extended for other obfuscated strings as well. Examples are the target telephone numbers of SMS messages, to aid existing pattern-based malware-detection tools. Injecting the action strings and URIs of Android intents used for inter-component and inter-application communication is also important. Many static analyses fail if these strings are not constant as they can no longer map intent

³This trick is adopted from the Booster component of the TamiFlex tool [6] for Java.

```

1 void obfuscated() {
2     String t = decodeMethodName();
3     this.class.getMethod(t).invoke(this);
4 }
5
6 void directCalls() {
7     String t = decodeMethodName();
8     if (t.equals("foo")) this.foo();
9     else if (t.equals("bar")) this.bar();
10    else this.class.getMethod(t).invoke(this);
11 }

```

Listing 10: Obfuscated Code Example

senders and receivers. The same applies to class-name strings used with explicit intents. If they are only decrypted at runtime, static analyses have no chance but to conservatively assume all possible recipients, which is highly imprecise. Injecting these strings as constants enables tools such as EPICC [31] to reconstruct the inter-component call graph more precisely and correctly identify the data flows between components and applications which would otherwise not be possible. Section V shows how enhancing apps with HARVESTER benefits TaintDroid and FlowDroid. We plan to assess further synergies in future work.

IV. IMPLEMENTATION

The static slicing is implemented using the Soot framework [44] for static program analysis and transformation, which can directly read and write Android APK files. HARVESTER uses Soot’s Jimple intermediate representation which is a flat, three-operand language optimized for static analyses.

To make the generated slices as precise (and thus as small) as possible, HARVESTER requires a precise call graph of the target application. Soot’s call-graph construction engine Spark provides such an initial call graph. Soot is capable of building a call-graph based on a program’s entry points. Since such an entry point does not exist for Android applications, an artificial main method is created using the `AndroidEntryPointCreator` component of FlowDroid [17], an existing Soot-based open-source static taint-analysis tool.

Initially, this graph is missing call edges for reflective calls, as exactly those calls are the task of HARVESTER to discover during its dynamic pass. Once the calls have been discovered, and embedded as direct calls in the APK, one can iterate the same procedure, expanding the call graph, potentially discovering more reflective call sites, dynamically resolving their targets, etc. In practice, however, we found that this reiteration is not usually necessary, as current malware does not usually call reflection APIs themselves using reflection.

HARVESTER executes the extracted slices by calling their entry points from an artificial executor activity injected into the resulting APK file. A slice, however, may have references to other Android components. Also, if the code was originally executed after the user clicks on a button, it may expect the hosting activity to be initialized. HARVESTER must thus emulate this activity’s lifecycle at runtime. Naive calls to lifecycle methods, however, can cause problems, as the Android operating system expects certain internal variables such as the used `Context` to be set which cannot be achieved through the normal interface. HARVESTER therefore uses reflection to inject these initialization values. The usual way of switching activities via `Intents` is not a suitable alternative as it would not give

HARVESTER control over the lifecycle methods, leading to unnecessary code execution.

Some entry points may raise unhandled exceptions, e.g., due to environment dependencies not supported by HARVESTER. Normally, in such a case the Android operating system would terminate the application, which would, however, also cancel the execution of all following paths. Therefore, HARVESTER explicitly catches and reports exceptions, but only skips the failing path and continues with the next one.

For very large programs, computing exact slices may be infeasible. HARVESTER therefore supports cut-offs that prevent it from walking further up (into callers) or down (into callees) along the call stack while slicing. After the cut-off, all further callees are taken as-is without any slicing. All callers exceeding the cut-off are simply disregarded, i.e., HARVESTER, assumes that the slice constructed so far does not depend on any earlier program logic. If some variables are not yet initialized, HARVESTER inserts artificial initialization statements that assign dummy values.

V. EVALUATION

We evaluated HARVESTER extensively, addressing the following five research questions:

- **Q1:** What is HARVESTER’s recall, i.e., how many logging points can it reach in real-world applications?
- **Q2:** How does the recall of HARVESTER relate to existing static- and dynamic-analysis approaches?
- **Q3:** How efficient is HARVESTER?
- **Q4:** Are there any interesting findings in apps?
- **Q5:** Can HARVESTER support other static/dynamic approaches to increase their recall?

The cut-offs for the caller- and callee-slicing were both set to 3 in all of our experiments. They can be configured as required, trading higher recall for longer analysis time.

Q1: What is HARVESTER’s recall?

We evaluated HARVESTER’s recall based on the coverage of logging points⁴. A perfect tool would reach every logging point. We chose the 13 different malware samples shown in Table I, some of which are known to be highly obfuscated (FakeInstaller, GinMaster and Obad). These samples heavily rely on reflection to mask the targets of method calls. Another malware family, Pincer, is known to hinder dynamic evaluation through anti-emulation techniques [33], [45]. Ssucl and Dougalek steal various private data items.

Table I shows the evaluation results for logging points from the categories *URI*, *Webview*, *SMS Number*, *SMS Text*, *File*, *Reflection* and *Shell Commands*. The results for each malware sample in each category are represented as circles. Grey slices indicate the fraction of logging points with constant values, where no backward-slicing and dynamic execution is necessary. Executing HARVESTER is not really necessary for those values,

⁴The reader be reminded that Section III-A defined a logging point as the combination of a statement and a value of interest

but nevertheless HARVESTER discovers those values at once. Green slices indicate the fraction of logging points with non-constant values for which HARVESTER was able to successfully retrieve at least one value. Red slices indicate the amount of missing logging points for which HARVESTER could not find a runtime value. This missed fraction is further represented on the right side of the circle. If a logging point is reachable along multiple control-flow paths, it may be executed with more than one runtime value. The number of distinct non-constant runtime values that HARVESTER extracts for the respective logging-point category is shown below the circle.

In summary, the table shows that, averaged over all categories, HARVESTER detects at least one value for 99% of all logging points. Due to the controlled execution of value-influencing branches, HARVESTER succeeds in reporting multiple values for many logging points. Discovering more than one value is useful for analyzing the behavior of an application (e.g., scamming telephone numbers for SMS fraud in different countries instead of just a single number). HARVESTER is even able to cope with the anti-analysis techniques used by the Pincer malware family where it successfully extracts the SMS number and message, URIs, shell commands and various file accesses.

The small fraction of missed logging points is mainly caused by missing initialization of important runtime values due to configured cut-offs during the static analysis (see Section III-A).

Q2: How does the recall of HARVESTER relate to existing static- and dynamic-analysis approaches?

In this section we compare HARVESTER with purely static and purely dynamic approaches for automatically detecting malicious applications.

Static Analysis: We compared HARVESTER with SAAF [20], a static approach for identifying parameter values based on a backward slicing approach starting from a method call. This method is similar to the static backward analysis part in HARVESTER (traditional slicing). SAAF and HARVESTER were evaluated on 2,600 malware samples from MobileSandbox [39]. The logging points for both tools were the number and the corresponding message of text messages. The results for SAAF show that the tool does not support semantics of string operations such as concatenation. Instead of the concatenated string, SAAF reports the two distinct operands. This gives only partial insight into the behavior of the application. In some cases⁵, SAAF did not even find all necessary fragments of the target telephone number (e.g. 1065-5021-80133). In contrast, HARVESTER extracts the final, complete SMS numbers for all of the samples and even reports numbers in cases in which SAAF did not yield any data.

Furthermore, SAAF does not support extracting the texts of the SMS messages being sent since they are usually not string constants, but built through concatenation and string transformation. Due to its static nature, SAAF cannot handle reflective calls either which is not an issue for HARVESTER due to the dynamic execution.

⁵e.g. sample MD5 b238628ff1263c0cd3f0c03e7be53bfd
















	URI	Webview	SMS Number	SMS Text	File	Reflection	Shell Cmd	Sum
FakeInstaller (MD5)								
b702b545d521f129e8efc1631a3abcee	 2	 5				 8,686		 8,691
dd40531493f53456c3b22ed0bf3e20ef						 1,686,341		 1,686,341
GinMaster (MD5)								
0d2deef5a03fc959c46550d6c2111c4a	 8				 17	 14		 39
ebe49b1b92a3b44eb159d15ca1f25c70	 9	 0			 4,486	 6		 4,501
Obad (MD5)								
58617e6a483f59bc93e500c65116eb87						 2,830		 2,830
e1064bfd836c4c895b569b2de4700284						 273		 273
dd1a3ff43330165298db703f7f0626ce						 214		 214
Pincer (MD5)								
b2b7d5999dce0559d13ab06d30c2c6ec	 1		 1	 1	 230		 1	 234
9c9afd6b77d8d3a66a2db2d2cf0b94b3	 3		 1	 1	 1,827	 0	 2	 1,834
Ssuel (MD5)								
f0bf007b3d2580297b208868425e98c7	 2		 1	 1			 0	 4
c5a2d14bc52f109a06641c1f15e90985	 0		 0	 0			 0	 0
Dougalek (MD5)								
95a04cfc5ed03c54d4749310ba29dda9	 0		 36	 36	 2			 74
91d57eb7ec2582e0600f21b08dac9538	 4							 4
SUMMARY	 29	 5	 39	 39	 6,562	 1,698,364	 3	 1,705,041

TABLE I: Precision-Evaluation of HARVESTER. Green slices: amount of logging points with non-constant values where a dynamic analysis is necessary for value extraction. Red slices: amount of missing logging points. Grey slices: amount of logging points with constant values where no static/dynamic analysis is necessary. Fraction next to circle: fraction of missing logging points for non-constant values. Value below circle: number concrete extracted values for non-constant logging points

This shows that hybrid approaches such as HARVESTER have the opportunity to handle semantic operations more effectively than purely static ones like SAAF.

Dynamic Analysis: As the results above show already, HARVESTER shows a higher recall than classical testing-based approaches currently known to literature [9], [21], [26], [36] which, on average, only reach about 30% to 60% code coverage. If one assumes that the logging points are uniformly distributed over the application code, this means that only 30% to 60% of all logging points are reached at all. These numbers suggest that a hybrid approach of slicing and executing is more appropriate to Android applications than finding concrete test patterns.

To validate this impression further, we also tested HARVESTER on 150 samples from 18 malware families taken from the Malware Genome Project [50]. We compared HARVESTER's recall with Google's Monkey [12] which we ran with at least 1,000 randomly-generated events per app that were limited to normal user interactions (click, swipe, navigation button use). The goal was to find the telephone numbers to which SMS messages are sent. To count the respective logging points reached by Monkey, we instrumented the bytecode of the malware samples to create a log entry directly before sending the message. The results were evaluated by using Logcat. All tests with Monkey were carried out on an Android 4.1 emulator (API version 16).

```

1 public void onStart(Intent intent, int i)
2 ContentResolver cr = getContentResolver();
3 Cursor contacts = cr.query(CONTENT_URI, null, ...);
4 SmsManager sms = SmsManager.getDefault();
5 if (cr.getCount() > 0) {
6     do {
7         int colIdx = cr.getColumnIndex("data1");
8         String telNo = cr.getString(int)>(colIdx);
9         sms.sendTextMessage(telNo, null, "I take
           pleasure in hurting small animals, just thought you
           should know that", ...);
10        } while (cr.moveToNext());
11        sms.sendTextMessage("73822", null, "text", ...);
12    }
13 }

```

Listing 11: “DogWars” Game from Malware Genome Project

As expected, the emulator-detection techniques prevented Monkey from ever reaching any logging points in most malware samples. In total, Monkey only found values for 15.6% of all logging points. In only 9.33% of all apps, it found a value for at least one logging point. As an example, Listing 11 shows malicious code extracted from the “DogWars” application. It accesses the user’s contact database in line 3. Only if contacts are available on the phone, see line 5, the app sends out the premium SMS message (line 11). When Monkey executes the application on an emulator, it usually finds the contact database to be empty and thus never reaches the logging point for sending SMS messages. As our results confirm, such behavior is common among modern malware applications. Since such checks, however, do not influence the target telephone number, HARVESTER simply removes the respective condition and correctly retrieves the number 73822. Note that the taunting text messages (line 9) get sent to every telephone number in the user’s address book and are thus data-dependent on the environment (i.e., the contact database). Thus no general value can be retrieved by any tool.

Many malicious applications such as the *GoldDream*, *BaseBridge*, and *BgServ* malware families as well as the *DogWars* app perform their malicious activities in a background service that is started when the phone is rebooted. To obtain the respective runtime values, traditional dynamic approaches must also generate such external events and restart the phone. HARVESTER instead directly executes the code slices containing the logging points and thus does not need to emulate these events.

Furthermore, tools such as Monkey can only improve code coverage by triggering interactions in the user interface. Some malware apps from the *GPSSMSSpy* family, however, contain a broadcast receiver that directly leaks incoming SMS messages and which is completely distinct from the UI. While Monkey never executes the respective code, HARVESTER directly invokes the slice containing the data leak regardless of its original position in the code.

To overcome these problems with Monkey, we used the AndroidHooker [7] open-source dynamic testing toolkit which first prepares the emulator with fake “personal user data” such as contacts, before installing the application and exercising it using Monkey. AndroidHooker also sends external events such as incoming SMS messages and reboots the emulator during the test to trigger actions that only happen at boot time. This approach was able to reveal the premium SMS message in the *DogWars* app, but does not solve the code-

coverage problem in general. For instance, it still fails if the malicious code is only executed after receiving a command from a remote server, such as in the *GoldDream* malware family. HARVESTER succeeds nevertheless, as the conditional checking for the server’s command is not part of the slice that HARVESTER computes, and the code containing the logging point is directly and unconditionally executed. Due to such problems, AndroidHooker only found 16.31% of all logging points. In 10.67% of all apps, it found a value for at least one logging point.

All in all, with finding values for 74.47% of all logging points HARVESTER shows a much better recall for these malware samples as it is limited neither by time bombs, nor logics bombs, and does not require any external inputs to be simulated. In 86% of all apps, a value for at least one logging point was found. The current recall seems to be lower than 100% only because of two reasons. First, HARVESTER currently does not handle inter-component communication. This feature can easily be added by integrating HARVESTER with the inter-component analysis tool EPICC [31], which we plan in future work. Since both tools are based on Soot, they should be directly compatible. The second reason is due to current technical limitations of Soot’s Dalvik frontend, which still fails to process some small fraction of apps.

Q3: How efficient is HARVESTER?

App Stores such as the Google Play Store receive many thousand new or updated Android apps per day [40] which they need to check for malicious behavior. Therefore, fast tools which scale to the size of the market are required. We tested HARVESTER on 10,282 malware samples from VirusShare [1], 620 apps (only SMS trojans) from the Malware Genome Project [50], and 2,600 malware samples from MobileSandbox [39]. We configured HARVESTER with 3 logging-points both for the SMS phone numbers and the respective text messages. 1,926 of all apps under evaluation contained at least one logging-point in each of these two categories. We focused on SMS numbers and messages since SMS trojans are among the most sophisticated malware applications today [15]. With HARVESTER, one can effectively find the real values for phone numbers and text messages and compare them to known blacklists or apply existing filters for identifying scamming malware.

The performance evaluations reported in this section were run on a computation server with 12 AMD Opteron 8356 cores running Debian Linux 2.6 with Oracle’s Java HotSpot 64-Bit Server VM version 1.7.0 and a maximum heap size of 20 GB to avoid intermediate garbage collection. We used the Android ARM emulator in version 22.6.0. On average, HARVESTER took between 25 and 45 seconds and it extracted various distinct SMS telephone numbers, and distinct SMS messages. This shows that HARVESTER can be used for mass analyses and delivers results very quickly.

Q4: Are there any interesting findings in apps?

In this section, we report interesting values that HARVESTER extracted from malware applications. Our analysis is based on the results from the previous section. Some of these results have already been found through earlier manual investigation

by security experts. However, to the best of our knowledge, HARVESTER is the first fully-automated approach that is able to discover all of these findings.

Hiding Sensitive Method Calls: A growing number of sophisticated Android malware applications such as Obad [42] uses reflection to call methods identified by encrypted string constants which only get decrypted at runtime. We used HARVESTER to recover the targets of these reflective method calls and found two popular obfuscation patterns. In the first pattern, only sensitive API calls, such as “getSubscriberID”, “getDeviceId”, or “sendTextMessage” are obfuscated, which is likely to be the result of a manual obfuscation to hinder human analysts or automatic tools that looks for sensitive API calls such as CHABADA [19]. In the second pattern, all method calls are obfuscated, even non-critical ones such as “StringBuffer.append()” or “String.indexOf()” which is most likely the result of automatic obfuscation tools such as DexGuard [41]. In some applications, even the reflective calls themselves were again called via reflection to produce a multi-stage obfuscation. The motivating example in Listing 1 is such a multi-stage obfuscation, which is very hard for a manual analyst to understand. HARVESTER is able to extract the called method as well as the concrete parameter values of the invocation in all these cases.

Premium-rate SMS and SMS Command and Control: Silently sending SMS messages to premium-rate numbers is one of the most common Android malware schemes [15]. Depending on the provider and the malware, a single message can cost from about 3.5\$ to 6\$ [8] which causes a high financial harm to the user. HARVESTER extracted many distinct premium-rate numbers from various known SMS trojan malware families such as “Pincer”. Many numbers can be found in multiple samples, making them good candidates for blacklisting. Since many samples are obfuscated, however, powerful extraction tools such as HARVESTER are required to reliably identify blacklisted numbers.

Furthermore, most SMS trojans store the number of messages sent in *SharedPreferences*, a key-value storage provided by the Android framework. HARVESTER finds many keys like “SENDED_SMS_COUNTER_KEY” or “sendCount” used for this purpose. Some samples even use keys like “cost” for storing the total amount of money stolen so far. Based on these values, the malware decides when the next premium-rate SMS message is sent. We also found applications that contact a command-and-control (C&C) server via SMS with commands such as “40659+3079+4128302+x+a”, or “3079+4128302+x+a”. Since the same commands reappear in many samples, they could also be used for blacklisting.

Interesting URIs: HARVESTER is able to extract the concrete URL of *http* requests sent by applications. These URLs can give hints as to whether an application is malicious or not. We extracted not only connections to advertisement servers, but also many well-known C&C server URLs⁶. Furthermore, it also extracts many interesting phone-local URIs for accessing content providers, such as “content://sms”, “content://mms” or “tel://<number>” which are used by malware for reading

⁶http://198.211.118.115:9081/Xq0jzoPa/g_L8jNgO.php, <http://m11g.net/q.php>, and others

SMS/MMS messages or initiating phone calls without user awareness [50]. In the case of the *tel* scheme, HARVESTER found the actual telephone numbers being called. In applications with advertisement libraries such as AirPush, HARVESTER revealed a lot of “market://details” URIs which open the PlayStore app to offer other apps for download.

Executed Commands: We also used HARVESTER to extract runtime values for command-executing API methods such as `Runtime.exec()`. Applications containing `su` and `chmod` commands are likely to be root exploits. HARVESTER can detect such commands even in the case of obfuscation.

Encryption Keys: Some benign applications encrypt sensitive data such as chat conversations, or credit card information, before storing it locally on the phone. This encryption, however, is rendered useless if the same hard-coded symmetric key is used for all installations of the app. Interestingly, this is the case in the popular WhatsApp messenger app [34]. Since the encrypted database is stored on the SD card, malicious applications can easily access it. Once the key is known, it can be decrypted and leaked. HARVESTER is able to fully automatically extract the WhatsApp encryption key by obtaining the values passed to the constructor of the `SecretKeySpec` class.

Q5: Can HARVESTER support other static/dynamic approaches to increase their recall?

Our results from Table I confirm that many applications use non-constant values which cannot be extracted by a simple forward scan. This is an issue since many static malware detection and classification approaches [39] rely, among others, on constant values such as hard-coded URLs. As HARVESTER extracts constant as well as non-constant values from apps, these approaches can be supported by HARVESTER’s output.

Static Analyses: We compared the recall of the FlowDroid [17] static data flow tracker on real-world malware applications before and after injecting runtime values for reflective method calls with HARVESTER (see Section III-C). We chose the Fakeinstaller.AH [37] malware family⁷ which is known for leaking private data, but also for its massive use of reflection to hide calls to sensitive API methods. On the original obfuscated sample, FlowDroid detected only 9 distinct leaks. After using HARVESTER with the option of replacing reflective calls with their respective actual callees, FlowDroid detected 26 privacy leaks, almost three times as many as before. These 26 leaks included stealing the IMEI or phone number via SMS, which was obfuscated through reflection. This also shows that HARVESTER was able to reconstruct correct values of interest for these reflective method calls.

Dynamic Analyses: Dynamic tools are known to suffer from limited code coverage if an application depends on user inputs. With the code slices extracted by HARVESTER, one can directly execute the code of interest, regardless of its original position in the original program. No user interaction with the application is required, eliminating code coverage issues with existing input generation approaches. In an approach similar

⁷Sample MD5: 38a9ed0b5577af6392096b4dc4a61e02

to Anubis [24], TaintDroid 4.1 was run inside the emulator on the Tapsnake [50] malware sample⁸ which steals location data only after a delay of 15 minutes [48]. On the original malware, the analyst needs to wait this time. With the app reduced by HARVESTER’s slicing approach, TaintDroid reports the leak instantly, without any UI interaction.

Precision and Recall: To evaluate how HARVESTER improves the precision and recall of existing tools on obfuscated applications, we tested FlowDroid and TaintDroid on ten randomly-picked applications from DroidBench [17] which we obfuscated using DexGuard [41]. All API method calls were replaced with reflective calls on encrypted strings. Table II compares the detection rate of FlowDroid and TaintDroid on the obfuscated applications without applying HARVESTER (*BEFORE* - column 2 and 4) to the respective detection rates after injecting the runtime values computed by HARVESTER (*AFTER* - column 3 and 5).

These results show that FlowDroid was initially not able to detect any leak in the obfuscated apps. After deobfuscating the apps with HARVESTER through runtime-value injection (see Section III-C), FlowDroid found the same leaks as in the unobfuscated original version. In “PrivacyDataLeak3”, FlowDroid always misses one of the two leaks, even in the original, unobfuscated file, for reasons unrelated to the work presented here.

TaintDroid was run *without any user interaction* with the respective apps. In the original app it thus missed leaks depending on user actions such as in “Button3”. Furthermore, TaintDroid originally failed on apps containing emulator-detection checks. When running the slices extracted by HARVESTER (the modified APK in Figure 1), both types of leaks are found fully automatically without any user or machine interaction. The remaining missing leaks occur due to TaintDroid not considering Android’s logging functions (e.g., `Log.i()`) as sinks, as we confirmed with the authors of TaintDroid.

These results show that HARVESTER can be used to improve the precision and recall of both, static as well as dynamic data flow analysis tools.

⁸Sample MD5: 7937c1ab615de0e71632fe9d59a259cf

⊗ = correct warning, ○ = missed leak
multiple circles in one row: multiple leaks expected

App (Obfuscated)	TaintDroid		FlowDroid	
	BEFORE	AFTER	BEFORE	AFTER
Enhancement				
Button1		⊗		⊗
Button3	○ ○	⊗ ○	○ ○	⊗ ⊗
FieldSensitivity3	⊗	⊗	○	⊗
ActivityLifecycle2	⊗	⊗	○	⊗
PrivateDataLeak3	⊗ ○	⊗ ○	○ ○	⊗ ○
StaticInitialization2	⊗	⊗	○	⊗
EmulatorDetection1	○ ○	⊗ ○	○ ○	⊗ ⊗
EmulatorDetection2	○ ○	⊗ ○	○ ○	⊗ ⊗
LoopExample1	⊗	⊗	○	⊗
Reflection1	⊗	⊗	○	⊗

TABLE II: Leak detection by TaintDroid and FlowDroid on Obfuscated DroidBench Apps before and after Value Injection / Slicing. Note that we did not have to interact with the app for the TaintDroid test.

VI. RELATED WORK

Researchers have proposed various approaches for analyzing the behavior of Android applications. Tools which simply convert the Android dex code back to Java source code such as dex2jar [14] or Dare [30] suffer from the problem that obfuscated applications do not contain sensitive values such as URLs or telephone numbers in plain, but the analyst rather needs to reconstruct them by manually applying the deobfuscation steps that would normally execute at runtime.

The remainder of this section describes more advanced approaches that provide a higher level of automation using static, dynamic, or hybrid analysis techniques.

Static Analysis: FlowDroid [17] is a static taint analysis tool which determines whether sensitive information is leaked in an Android application. FlowDroid inherits basic support for reflective method calls from the Soot framework on which it is based, but cannot handle cases in which the string containing the target class or method name is decrypted or concatenated dynamically at runtime. CHEX [25] is a tool that detects component hijacking vulnerabilities in Android applications by tracking taints between externally accessible interfaces and sensitive sources or sinks. Just like FlowDroid, the approach relies on a complete call graph and thus fails if call targets are obfuscated using reflection. Therefore, CHEX would also benefit from our runtime value injection for a more complete analysis. SAAF [20] is a purely static tool for finding constant strings in Android applications based on backwards slicing. It does not aim at providing any runtime semantics, e.g., if an application decrypts a constant string at runtime, SAAF will only produce the original ciphertext, leaving substantial work with the human analyst.

Dynamic Analysis: Dynamic approaches that profile runtime behavior such as Google Bouncer [29] can only detect runtime values that violate the Play Store’s policy (e.g., blacklisted URLs or telephone numbers) if they are actually used in API calls during the test run. Malware, however, often employs sophisticated mechanisms to detect whether it is run in an emulator or simply waits for longer than the test run lasts before starting the malicious behavior. TaintDroid [13] is a dynamic data-flow tracker which detects leaks of sensitive information at runtime. Other techniques such as Aurasium [47] inject a native code layer between the operating system and the Android application which intercepts sensitive API calls and checks the data passed to them. All these approaches share the problem of only finding values in code that is actually executed, thus requiring a test driver with full code coverage. HARVESTER circumvents this problem by directly executing the code of interest regardless of its position in the original application.

Hybrid Analysis: TamiFlex [6] monitors reflective method calls in Java applications at runtime and injects the found call targets into the application as call edges to aid static analysis tools. It however does not support Android and employs no slicing. Instead, it always executes the application as a whole, leaving open how full coverage of callees is to be achieved during the runtime analysis part. AppDoctor [22] slices Android applications to find user interactions that lead to application crashes. AppDoctor’s hybrid slice-and-run principle

is similar to HARVESTER. However, AppDoctor executes the complete derived UI actions, while HARVESTER's slices only contain code contributing to the value of a concrete value of interest. AppSealer [49] performs static taint tracking on an Android application and then instruments the app along the respective propagation paths to monitor for actual leaks at runtime, effectively ruling out false positives introduced by the static analysis. It then fixes component-hijacking vulnerabilities at runtime if sensitive data reaches a sink. This approach can, however, not find leaks missed by the static analysis and thus inherits the problem of reflective method calls. SMV-Hunter [38] scans for custom implementations of the SSL certificate validation in Android applications. It first statically checks whether custom validation routines are present. If so, the dynamic part attempts to trigger this code and confirm a man-in-the-middle vulnerability. The tool only supports simple UI interactions that neither span multiple pages nor require complex inputs.

UI-Automation: SwiftHand [9] uses machine-learning to infer a model of the application which is then used to generate concrete input sequences that visit previously unexplored states of the app. SwiftHand however has limited code coverage; on complex user interfaces coverage can fall under 40%. Code that is only executed in specific environments (e.g., depending on data loaded from the Internet) might not be reached at all. Dynodroid [26] instruments the Android framework for capturing events from unmodified applications, generated both by automatic techniques such as MonkeyRunner [12] and by human analysts. On average, it achieves a code coverage of 55%. AppsPlayground [36] uses an enhanced version of TaintDroid [13] for dynamic data flow tracking. The authors changed the Android framework to additionally monitor specific API and kernel level methods. For exercising the application at runtime, they used random testing guided by heuristics leading to a code coverage of about 33%. Applications that rely on user input such as credentials may not run correctly and malware can evade analysis by detecting the emulator and refraining from its malicious behavior.

Since HARVESTER directly executes the code fragments of interest, dependencies on user input as well as emulator detection code that does not influence the actual runtime values of interest are removed in our approach, avoiding these problems of existing techniques.

VII CONCLUSIONS

In this paper, we have presented HARVESTER, a novel hybrid approach for extracting runtime values from Android applications even in the case of obfuscation and powerful anti-analysis techniques (e.g., emulator detection, time bombs or logic bombs). We have shown that HARVESTER can be used as a deobfuscator and finds, among others, plain-text telephone numbers of SMS trojans, command and control messages of bots, and reflective call targets of various types of malware. HARVESTER yields a far better coverage of logging points than current state-of-the-art UI automation approaches. We have evaluated HARVESTER both as a standalone tool and as an aid for existing static and dynamic analyses by enhancing applications with the deobfuscated runtime values. Our results show that HARVESTER massively improves the recall of current

state-of-the-art static and dynamic data flow analysis tools. On average, HARVESTER analyzes an application in less than one minute, yielding dynamically-computed runtime values which could not be retrieved with existing approaches.

Acknowledgements: This work is supported by a Google Faculty Research Award, by the BMBF within EC SPRIDE, by the Hessian LOEWE excellence initiative within CASED, and by the DFG's Priority Program 1496 Reliably Secure Software Systems and the project RUNSECURE.

REFERENCES

- [1] Virus share, aug 2013. <http://virusshare.com/>.
- [2] PF Adam, A Chaudhuri, and JS Foster. Scandroid: Automated security certification of android applications. In *IEEE symposium of security and privacy*, 2009.
- [3] Android Central. Larry page: 1.5 million android devices activated every day, July 2013. <http://www.androidcentral.com/larry-page-15-million-android-devices-activated-every-day>.
- [4] Jean-Francois Bergeretti and Bernard A. Carré. Information-flow and data-flow analysis of while-programs. *ACM Trans. Program. Lang. Syst.*, 7(1):37–61, January 1985.
- [5] Thomas Bläsing, Aubrey-Derrick Schmidt, Leonid Batyuk, Seyit A. Camtepe, and Sahin Albayrak. An android application sandbox system for suspicious software detection. In *5th International Conference on Malicious and Unwanted Software (Malware 2010) (MALWARE'2010)*, Nancy, France, France.
- [6] Eric Bodden, Andreas Sewe, Jan Sinschek, Mira Mezini, and Hela Oueslati. Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders. In *Proceedings of the 33rd International Conference on Software Engineering, ICSE '11*, pages 241–250, New York, NY, USA, 2011. ACM.
- [7] Georges Bossert and Dimitri Kirchner. How to play hooker: Une solution d'analyse automatisée de markets android.
- [8] Eric Chien. Motivations of recent android malware, 2012. http://investor.symantec.com/files/doc_news/2012/motivations_of_recent_android_malware.pdf.
- [9] Wontae Choi, George Necula, and Koushik Sen. Guided gui testing of android apps with minimal restart and approximate learning. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages: Applications, OOPSLA '13*, pages 623–640, New York, NY, USA, 2013. ACM.
- [10] Kevin Coogan, Saumya Debray, Tasneem Kaochar, and Gregg Townsend. Automatic static unpacking of malware binaries. In *Proceedings of the 2009 16th Working Conference on Reverse Engineering, WCRE '09*, pages 167–176, Washington, DC, USA, 2009. IEEE Computer Society.
- [11] Luis Corrons. New malware attack through google play. Website of Pandalabs, Feb 2014. <http://pandalabs.pandasecurity.com/new-malware-attack-through-google-play/>.
- [12] Google Developers. monkeyrunner. Website of Google Developers, Mai 2014. http://developer.android.com/tools/help/monkeyrunner_concepts.html.
- [13] William Enck, Peter Gilbert, Byung gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol Sheth. Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *OSDI*, pages 393–407, 2010.
- [14] William Enck, Damien Oetean, Patrick McDaniel, and Swarat Chaudhuri. A study of android application security. In *Proceedings of the 20th USENIX Conference on Security, SEC'11*, pages 21–21, Berkeley, CA, USA, 2011. USENIX Association.
- [15] F-Secure. Mobile threat report q1 2014, April 2014. http://www.f-secure.com/static/doc/labs_global/Research/Mobile_Threat_Report_Q1_2014_print.pdf.
- [16] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for java. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation, PLDI '02*, pages 234–245, New York, NY, USA, 2002. ACM.

- [17] Christian Fritz, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves le Traon, Damien Oceau, and Patrick 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 (PLDI)*, June 2014.
- [18] Google Inc. Proguard. Android Developer Website. <http://developer.android.com/tools/help/proguard.html>.
- [19] Alessandra Gorla, Ilaria Tavecchia, Florian Gross, and Andreas Zeller. Checking app behavior against app descriptions. In *ICSE'14: Proceedings of the 36th International Conference on Software Engineering*, 2014.
- [20] Johannes Hoffmann, Martin Ussath, Thorsten Holz, and Michael Spreitzenbarth. Slicing droids: Program slicing for smali code. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, SAC '13, pages 1844–1851, New York, NY, USA, 2013. ACM.
- [21] Cuixiong Hu and Iulian Neamtii. Automating gui testing for android applications. In *Proceedings of the 6th International Workshop on Automation of Software Test*, AST '11, pages 77–83, New York, NY, USA, 2011. ACM.
- [22] Gang Hu, Xinhao Yuan, Yang Tang, and Junfeng Yang. Efficiently, effectively detecting mobile app bugs with appdoctor. *EuroSys*, 2014.
- [23] International Data Corporation. Worldwide quarterly mobile phone tracker 3q12, November 2012. http://www.idc.com/tracker/showproductinfo.jsp?prod_id=37.
- [24] International Secure Systems Lab. Anubis - malware analysis for unknown binaries. Website of Anubis, mai 2014. <http://anubis.iseclab.org>.
- [25] Long Lu, Zhichun Li, Zhenyu Wu, Wenke Lee, and Guofei Jiang. Chex: statically vetting android apps for component hijacking vulnerabilities. In *CCS 2012*, pages 229–240, 2012.
- [26] Aravind Machiry, Rohan Tahiliani, and Mayur Naik. Dynodroid: An input generation system for android apps. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 224–234, New York, NY, USA, 2013. ACM.
- [27] Private communication with researchers from mcafee, 11 2014.
- [28] Daisuke Nakajima. Vietnamese adult apps on google play open gate to sms trojans. Website of McAfee Labs, Jan 2014. <http://blogs.mcafee.com/mcafee-labs/vietnamese-adult-apps-google-play-open-gate-to-sms-trojan>.
- [29] J Oberheide and C Miller. Dissecting the android bouncer. *SummerCon2012, New York*, 2012.
- [30] Damien Oceau, Somesh Jha, and Patrick McDaniel. Retargeting android applications to java bytecode. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, pages 6:1–6:11, New York, NY, USA, 2012. ACM.
- [31] Damien Oceau, Patrick McDaniel, Somesh Jha, Alexandre Bartel, Eric Bodden, Jacques Klein, and Yves Le Traon. Effective inter-component communication mapping in android with epicc: An essential step towards holistic security analysis. In *Proceedings of the 22Nd USENIX Conference on Security*, SEC'13, pages 543–558, Berkeley, CA, USA, 2013. USENIX Association.
- [32] Karl J. Ottenstein and Linda M. Ottenstein. The program dependence graph in a software development environment. *SIGPLAN Not.*, 19(5):177–184, April 1984.
- [33] Thanasis Petsas, Giannis Voyatzis, Elias Athanasopoulos, Michalis Polychronakis, and Sotiris Ioannidis. Rage against the virtual machine: Hindering dynamic analysis of android malware. In *Proceedings of the Seventh European Workshop on System Security*, EuroSec '14, pages 5:1–5:6, New York, NY, USA, 2014. ACM.
- [34] Google Play. Whatsapp messenger. Website of Google PlayStore, Mai 2014. <https://play.google.com/store/apps/details?id=com.whatsapp>.
- [35] Thomas Raffetseder, Christopher Kruegel, and Engin Kirda. Detecting system emulators. In *Proceedings of the 10th International Conference on Information Security*, ISC'07, pages 1–18, Berlin, Heidelberg, 2007. Springer-Verlag.
- [36] Vaibhav Rastogi, Yan Chen, and William Enck. Appsplayground: Automatic security analysis of smartphone applications. In *Proceedings of the Third ACM Conference on Data and Application Security and Privacy*, CODASPY '13, pages 209–220, New York, NY, USA, 2013. ACM.
- [37] Fernando Ruiz. Fakeinstaller leads the attack on android phones. Website of McAfee Labs, Oct 2012. <https://blogs.mcafee.com/mcafee-labs/fakeinstaller-leads-the-attack-on-android-phones>.
- [38] David Sounthiraraj, Justin Sahs, Garrett Greenwood, Zhiqiang Lin, and Latifur Khan. Smv-hunter: Large scale, automated detection of ssl/tls man-in-the-middle vulnerabilities in android apps. In *Proceedings of the 21st Annual Network and Distributed System Security Symposium (NDSS'14)*, San Diego, CA, February 2014.
- [39] Michael Spreitzenbarth, Felix Freiling, Florian Echter, Thomas Schreck, and Johannes Hoffmann. Mobile-sandbox: Having a deeper look into android applications. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, SAC '13, pages 1808–1815, New York, NY, USA, 2013. ACM.
- [40] AppBrain Stats. Number of android applications. Android Statistics Page of AppBrain, March 2014. <http://www.appbrain.com/stats/number-of-android-apps>.
- [41] Saikoa Applied Compiler Technology. Dexguard. Website of Saikoa, Feb 2014. <http://www.saikoa.com/dexguard>.
- [42] Emre Tinaztepe, Doğan Kurt, and Alp Güleç. Android obad. Technical report, COMODO, July 2013.
- [43] Frank Tip. A survey of program slicing techniques. Technical report, Amsterdam, The Netherlands, The Netherlands, 1994.
- [44] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot - a java bytecode optimization framework. In *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research*, CASCON '99, pages 13–. IBM Press, 1999.
- [45] T. Vidas and N. Christin. Evading Android runtime analysis via sandbox detection. In *Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security (ASIACCS'14)*, Kyoto, Japan, June 2014.
- [46] Mark Weiser. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering*, ICSE '81, pages 439–449, Piscataway, NJ, USA, 1981. IEEE Press.
- [47] Rubin Xu, Hassen Saïdi, and Ross Anderson. Aurasium: practical policy enforcement for android applications. In *USENIX Security 2012*, Security'12, 2012.
- [48] Zheming Yang, Min Yang, Yuan Zhang, Guofei Gu, Peng Ning, and X. Sean Wang. Appintert: analyzing sensitive data transmission in android for privacy leakage detection. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, CCS '13, pages 1043–1054, New York, NY, USA, 2013. ACM.
- [49] Mu Zhang and Heng Yin. Appsealer: Automatic generation of vulnerability-specific patches for preventing component hijacking attacks in android applications. In *Proceedings of the 21st Annual Network and Distributed System Security Symposium (NDSS'14)*, February 2014.
- [50] Yajin Zhou and Xuxian Jiang. Dissecting android malware: Characterization and evolution. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, SP '12, pages 95–109, Washington, DC, USA, 2012. IEEE Computer Society.