Harvesting Runtime Values in Android Applications That Feature Anti-Analysis Techniques

Siegfried Rasthofer, Steven Arzt, Marc Miltenberger
Secure Software Engineering Group
Technische Universität Darmstadt &
Fraunhofer SIT, Darmstadt, Germany
{siegfried.rasthofer, steven.arzt, marc.miltenberger}@cased.de

Eric Bodden*
Software Engineering Group
University of Paderborn &
Fraunhofer IEM, Paderborn, Germany
eric.bodden@uni-paderborn.de

***Abstract*—It is generally challenging to tell apart malware from benign applications. To make this decision, human analysts are frequently interested in runtime values: targets of reflective method calls, URLs to which data is sent, target telephone numbers of SMS messages, and many more. However, obfuscation and string encryption, used by malware as well as goodware, often not only render human inspections, but also 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 we therefore present HARVESTER, an approach to fully automatically extract runtime values from Android applications. HARVESTER is designed to extract values even from highly obfuscated state-of-the-art malware samples that obfuscate method calls using reflection, hide sensitive values in native code, load code dynamically and apply anti-analysis techniques. The approach combines program slicing with code generation and dynamic execution.

Experiments on 16,799 current malware samples show that HARVESTER fully automatically extracts many sensitive values, with perfect precision. The process usually takes less than three minutes and does not require human interaction. In particular, it goes without simulating UI inputs. Two case studies further show that by integrating the extracted values back into the app, HARVESTER can increase the recall of existing static and dynamic analysis tools such as FlowDroid and TaintDroid.

I. INTRODUCTION

To assess the quality or security of a mobile application, experts are frequently interested in its runtime values. For instance, the analyst often needs to know which method a reflective method call is invoking, which URL a piece of 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.). Even in benign applications runtime values are hard to extract precisely, but modern malware such as Pincer, Obad [1] or FakeInstaller [2] creates an even greater challenge by obfuscating runtime values deliberately. The malware stores such values (e.g., reflective call targets, the target telephone numbers of SMS scams, or the addresses of remote command&control servers) in an encrypted format inside the application code, to be decrypted only at runtime.

```
1 if(Build.FINGERPRINT.startsWith("generic"))
2   return;  //we are running in an emulator
3 String messageText = simCountryIso().equals("US") ? US
      : INTERN;
4 String clazz = decrypt("fri$ds\&S");
5 String method = decrypt("dvdf4$DCS");
6 Class.forName(clazz).
7   getMethod(method).invoke(
      "+01234",null,messageText,null,null);
```

Listing 1: Simplified Example

Listing 1 shows a simplified example, inspired by the Pincer malware [3]. In lines 6–7, it sends an SMS message to a phone number. Manually deducing the target of this obfuscated call is time-consuming and tedious as the analyst needs to first understand and reconstruct the decryption routine `decrypt` to obtain the actual runtime arguments of the call. Only then, the analyst knows that the reflective call references the `SmsManager` class (line 4) and its `sendTextMessage` method (line 5).

This obfuscation technique not only raises the bar for human analysts, it also effectively hinders all current static analysis approaches. Many current static analyses either do not handle reflection at all or only support constant target strings [4]–[6]. Therefore, they would be unable to detect that the example sends an SMS message at all. Other approaches model the String API to find reflective call targets [7]. In the example, however, these approaches will likely not be able to correctly interpret the `decrypt` function, especially if implemented in native code. Consequently, those approaches would miss the SMS message as well. In general, static analyses will always have an incomplete picture of the code's behavior, because their handling of runtime values can never be complete—ultimately due to the halting problem.

If static analysis fails one might think that maybe dynamic analysis can come to the rescue. Current malware, however, also fools dynamic analyses. This is because many malicious applications nowadays contain so-called *time bombs* or *logic bombs* [8]–[10]. Logic bombs cause an app to suppress any malicious activity if the app itself detects that it is executing within an analysis environment [11]. Time bombs cause an app

*At the time this research was conducted, Eric Bodden was employed at Fraunhofer SIT & TU Darmstadt.

NDSS '16, 21-24 February 2016, San Diego, CA, USA
 ISBN 1-891562-41-X
http://dx.doi.org/10.14722/ndss.2016.23066

to suppress the malicious behavior in any case for a longer period of time, or until after a reboot of the phone, etc. This also includes botnet malware that only acts in response to a command received from a command-and-control server—a command that dynamic analysis tools will find virtually impossible to guess correctly. Moreover, for all applications, including benign ones, a dynamic analysis can only reason about code paths that the analysis actually executes. However, neither an automatic event-generation or UI-testing tool, nor a human analyst can generally cover all possible execution paths in a finite amount of time, causing most dynamic analyses to be incomplete. Even current approaches [12] do not yet scale very well and can take hours even for medium-sized apps.

In this work we present HARVESTER, a novel approach that seeks to effectively address all of the above problems for current malware samples. Even for the most sophisticated malware families such as Obad, Pincer, or FakeInstaller, HARVESTER is able to extract virtually all runtime values of interest within minutes, without any user intervention, and in our experiments with perfect accuracy. The tool, soon to be integrated into a commercial product, works fully automatically. The analyst only needs to provide an Android app's binary code and a configuration file naming the code locations at which HARVESTER should extract values. HARVESTER works through a particular variation of traditional static-analysis algorithms known from program slicing combined with code generation and concrete dynamic code execution.

Our evaluation on 16,799 current malware samples shows that HARVESTER discovers values for 86,6% of all requests. In particular, for the current malware samples tested, HARVESTER can completely resolve the targets of encoded reflective method calls in almost every single case. For a representative subset of samples, we manually verified that HARVESTER shows a precision and recall of 100% for extracting SMS messages, SMS numbers and shell commands. Furthermore, on average HARVESTER takes less than three minutes per app to extract concrete telephone numbers and text messages from a large number of potential SMS trojans. During our 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 reflection. Moreover, HARVESTER successfully extracts the obfuscated key used by the well-known benign WhatsApp messaging app [13] to encrypt its message store.

In addition to the above evaluation, in this paper we also present two case studies that showcase further important example applications of HARVESTER's results. We explain how HARVESTER can improve the coverage of existing off-the-shelf static as well as dynamic analysis tools. As the first case study shows, static-analysis tools such as FlowDroid [4] can greatly improve their recall, i.e., find more data leaks, when incorporating reflection information computed by HARVESTER. The second case study reveals that even dynamic analysis tools such as TaintDroid [14] can benefit greatly from HARVESTER, as an integration with HARVESTER allows the dynamic analysis to effectively circumvent time and logic bombs and thus to find otherwise dormant malware.

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

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

- a variation of traditional slicing algorithms fine-tuned to support the hybrid extraction of runtime values in Android applications,
- a dynamic execution system for running the computed code slices and extracting the values of interest without user interaction,
- an evaluation of the approach's feasibility for a mass-analysis on real-world malware applications, and
- two case studies assessing how HARVESTER can improve the coverage of existing off-the-shelf static and dynamic analysis tools.

The remainder of this paper is structured as follows: Section II gives a more detailed example of current, obfuscated Android malware, Section III explains how HARVESTER is configured and Section IV gives a high-level overview. In Section V, we explain HARVESTER's architecture and the algorithms used to compute the runtime values in detail. Section VI reports on our experimental evaluation, and Section VII reports on two case studies. Further use cases of HARVESTER are discussed in Section VIII and potential future attacks on HARVESTER are discussed in Section IX, while Section X gives an overview of related work and Section XI concludes the paper.

II. MOTIVATING EXAMPLE

Listing 2 shows a real-world code snippet taken from FakeInstaller [2][1], one of the most widespread malware families [15]. To ease understanding, we decompiled the sample to Java source code and added comments to the code. The malware authors obfuscated their app with random class and method names, eliminating most semantic information. Furthermore, FakeInstaller heavily relies on obfuscation to hide its behavior from both analysis tools and manual investigators. 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.

Many current malware applications are obfuscated in a similar way, either manually or by using commercial tools such as DexGuard [16]. For a human analyst to understand the runtime behavior of such obfuscated code, she must know the target methods of the reflective calls. In the example, these values are the decoded class name in line 6 and the decoded method name in line 9. To find these values manually, she would have to carefully inspect the decompiled bytecode, find the lookup table, and manually decrypt all strings to detect the malicious behavior. Strings decrypted for one application once cannot usually be reused, as different malware variants use different lookup tables.

Static code-analysis approaches such as SAAF [17] apply techniques such as backward slicing in order to extract

[1]Sample MD5: dd40531493f53456c3b22ed0bf3e20ef

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

Listing 2: Highly obfuscated code sending a text message (FakeInstaller [2] malware family)

constant string information. These tools, however, have well-known limitations that make them fail on highly obfuscated applications, e.g., ones with dynamically-computed values as shown in Listing 2. Even those static-analysis tools that model the full string API still have limitations that can easily be exploited by malware developers. For example, one can implement the string-decoding method in a custom library written in native code. To the best of our knowledge, no static analysis tool for Android supports such native code.

The code in the example challenges dynamic analysis approaches as well. The analyses first have to find an execution path actually triggering the `gdadbjrj` method. If, for instance, method `gdadbjrj` is only executed after a delay or after a specific environment trigger, then this is a non-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. The analysis tool must then first perform all user actions required to reach the user-interface state displaying the button. Afterwards, it must be able to emulate this button click.

Additionally, various anti-analysis techniques for dynamic approaches, such as emulator-detection mechanisms [9], [10], [18] 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 [8] and thus fail on sophisticated malware.

HARVESTER, on the other hand, fully automatically retrieves all relevant runtime values of the example in Listing 2. The security analyst simply specifies the variables for which runtime values should be retrieved. For the example, we assume that the security analyst knows that she is interested in the parameters given to any calls to `SmsManager.sendTextMessage` that the application may make. As one can easily see, the code in Listing 2 contains no direct call to this API. Instead, the calls to this API are issued through reflection. But HARVESTER comes pre-configured with a setting that further extracts the parameters to such reflective calls, and inlines calls accordingly, once discovered.

In a first step, HARVESTER would hence attempt to extract parameters to the `forName` (line 11) and the `getMethod` calls (line 15). HARVESTER's static slicer 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. All code outside of `gdadbjrj` is removed as it is not necessary for computing the values in question. 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.sendTextMessage` of the method called through reflection. In result, it replaces the original reflective method call by a direct call to that very API, and re-iterates the extraction process.

Assuming that the security analyst configured HARVESTER to extract the arguments given to such calls, HARVESTER performs a slicing for `paramString1` and `paramString2`. Once again, the emulator check is removed, but this time, the caller of method `gdadbjrj` must also be analyzed. In this caller, HARVESTER keeps exactly the code that computes the input values `paramString1` and `paramString2`. Afterwards, in the same way as before, the reduced code is run, and HARVESTER 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-bytecode transformation.

III. LOGGING POINTS AND VALUES OF INTEREST

The main purpose of HARVESTER is to compute runtime values. Formally we call these runtime values *values of interest*. To use HARVESTER, a human analyst defines *logging points* for which she wants to extract all values of interest. Both are defined as follows.

Definition 1: A *logging point* $\langle$v, s$\rangle$ comprises a variable or field access v and a statement s such that v is in scope at s.

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

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 *sendTextMessage* method such as `s: sendTextMessage(targetNumber, arg2, messageText, arg4, arg5)` where $\langle$ `targetNumber, s`$\rangle$ and $\langle$ `messageText, s`$\rangle$ are possible logging points at `s`. Parameters `arg2`, `arg4` and `arg5` can be also defined as logging points, but do not provide security-relevant information. The corresponding runtime values are the values of interest. Examples for values of interest for `targetNumber` would be `'+01234'` and for `messageText` would be `'This is a premium SMS message'`.

To ease the definition of logging points for the human analyst, HARVESTER provides a comprehensive list of pre-

defined logging points taken from SuSi [19], a machine-learning approach which provides a comprehensive list of categorized sensitive API methods. HARVESTER makes use of these sensitive API methods by providing generic categories such as *URL*, *Shell-Command* or *SMS Number* as tool-input parameters. For instance, if one is interested in URLs inside the application, one can run HARVESTER with the *URL* parameter and all API calls that are able to call a URL are automatically defined as logging points. This is the only human interaction that HARVESTER requires.

IV. Overall Approach

Figure 1 depicts HARVESTER's workflow. To compute values of interest, HARVESTER first reads the APK file and a configuration file defining the logging points. HARVESTER next computes a static backwards slice starting at these code points, as will be further explained in Section V-A. This slicing step runs on a desktop computer or compute 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.

HARVESTER additionally alters those conditionals whose value depends on the execution environment and on which the slicing criterion, i.e., the value in question, is data-dependent. These conditionals are replaced by simple Boolean variables, allowing HARVESTER to force the simulation of different environments at runtime. Listing 3 shows the output of Harvester when requesting a slice for the parameters of the reflective call from Listing 1. The emulator check has been removed, as the slicing criterion is reachable only if the branch falls through. The conditional in line 3 has been replaced by the global variable `EXECUTOR_1`, making the slice parametric.

This new, reduced APK file is then executed on a stock Android emulator or real phone, as we explain in Section V-B. These steps are fully automated and no user interaction is required. In a *forced execution*, HARVESTER explicitly triggers all the different behaviors of the parametric slice (in Listing 3 with both `true` and `false` for `EXECUTOR_1`) which allows the complete reconstruction of the values of interest, for all concrete environments, decrypting any encrypted values. HARVESTER instruments the reporting mechanism for the values of interest into the slices (see line 4), making changes to the runtime environment (emulator, Android OS) unnecessary. Note that HARVESTER does not need to reconfigure or reset the actual device or emulator on which the slices are executed which is novel in comparison to other approaches that are based on symbolic or concolic execution [20], [21].

In some highly-obfuscated applications, the logging point cannot directly be identified. Reconsider Listing 2, in which the application uses reflection to call the method that sends out a text message. To handle such cases, HARVESTER can

```
1 String messageText = EXECUTOR_1 ? US:INTERN;
2 String clazz = decrypt("fri$ds\&S");
3 String method = decrypt("dvdf4$DCS");
4 Harvester.report(clazz,method,messageText);
5 Class.forName(clazz).
6   getMethod(method).invoke(
      "+01234",null,messageText,null,null);
```

Listing 3: Sliced version of Listing 1

be configured to run twice, once to retrieve the targets of all reflective method calls, and then a second time to retrieve the telephone number and/or text messages for the reflectively invoked calls to the SMS interface just identified.

V. Detailed Solution Architecture

Next we provide more details about the main components of HARVESTER namely the *static backward slicing* process, the *dynamic execution of the reduced APK* and the *injection of runtime values into the APK*, as shown in Figure 1.

A. Static Backward Slicing

Part A comprises the *static analysis* phase. In traditional slicing as defined by Weiser [22], a program slice S is an *executable program* that is obtained from a program P by removing statements such that S replicates the behavior of P [23] with respect to the so-called *slicing criterion*—a value of interest selected by the user. We use Figure 2 to explain the effect of traditional slicing on our initial example from Listing 1. Assume that we want to slice this program such that the parameters `clazz`, `method` and `messageText` passed to the reflective final call are our slicing criteria. The reflective call is data-dependent on all four assignments to those three variables. The assignments to `messageText` are further control-dependent on the check of the `simCountryIso()`. All of those statements are further control-dependent on the check on `Build.FINGERPRINT`, the environment check that circumvents the execution of the remaining code on Android emulators. Traditional slicing approaches such as the one by Weiser [22] would include this check in the slice. Executing the check, however, immediately leads to leaving the method, consequently never triggering the "interesting code" that computes the values relevant to the slicing criterion.

Even if the emulator check were removed, this traditional approach would still not be sufficient as it would still retain the environment-dependent check on `simCountryIso()`. In the specific scenario of malware analysis, the method `simCountryIso()` will return exactly one of several country codes, depending on the configuration of the emulator. But frequently, the malware analyst is interested in inspecting *all possible* runtime values in question. In the example, we would like to cover both possible branches. Without further extensions to the approach this would require a reboot and reconfiguration of the emulator, which is a time consuming and error-prone undertaking. However, while the assignments to `msg` are *control*-dependent on `simCountryIso()` and thus also on the execution environment, there is no *data* dependency. HARVESTER exploits this fact by replacing the conditional referring to `simCountryIso()` by a simple global Boolean flag `EXECUTOR_1`. This flag causes the slice to become *parametric*: the selection of any concrete Boolean values for the generated control variables allows the direct execution of one of the parametric slices. This effectively breaks the dependencies of the app's execution on its execution environment, depicted by the lower red cross in Figure 2.

The same concept also applies to the dependency on `Build.FINGERPRINT`. In this case, the code of interest is, however, only executed if this check returns `true`. In other cases, the whole computation of the values of interest would be skipped. Therefore, this conditional is replaced by `true`, resulting in a removal of the condition as shown at the big red cross.

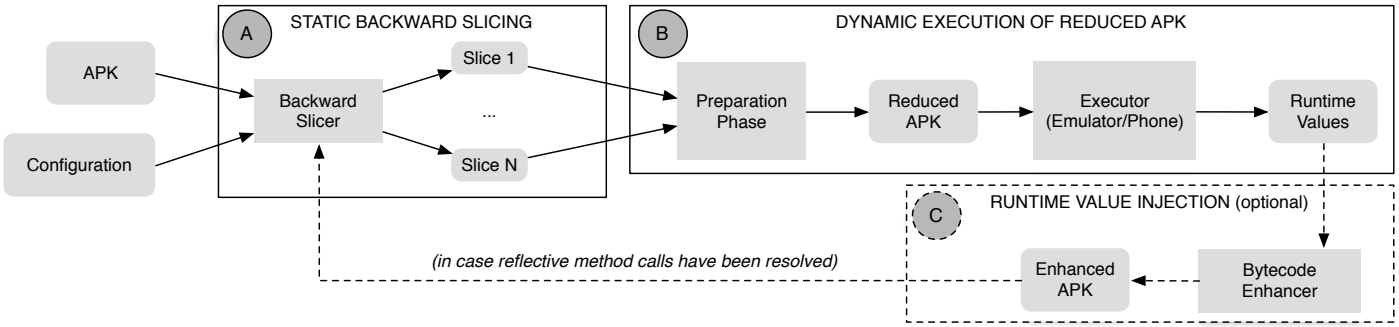

Fig. 1: Workflow of HARVESTER

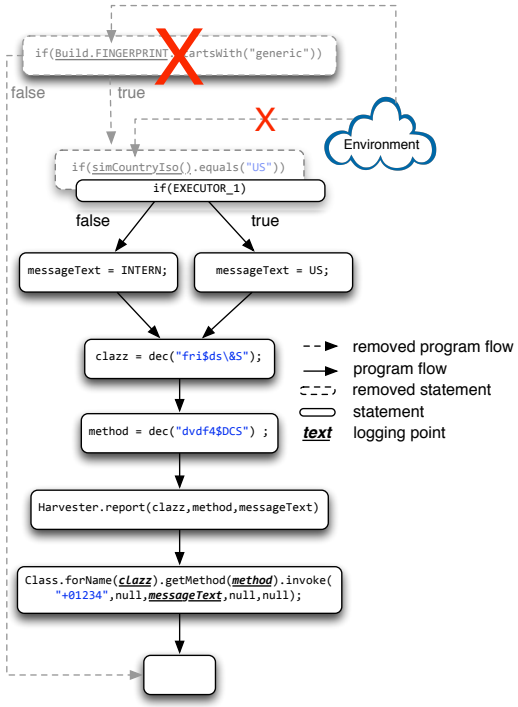

Fig. 2: Slice representation of Listing 1

Note that HARVESTER only parameterizes the slice at those conditionals that are data-dependent on environment values, while all other conditionals remain unchanged. This, for instance, allows HARVESTER to swiftly recover the correct value 123 for the `valueOfInterest` in the following example, which contains no such reference. (Note that in this snippet we show a `goto` operation. HARVESTER works directly on the bytecode level, where all loops are expressed that way.)

```
int valueOfInterest = 120, i = 0;

label1: if (i < 3) {
  i++;
  valueOfInterest++;
  goto label1;
}

send("" + valueOfInterest, "Hello");
```

If HARVESTER were to replace all conditionals regardless of whether they are environment-dependent or not, the slice for the example above would compute the incorrect value 120 when choosing `false` for the condition $i < 3$. Worse, when choosing `true` for the condition, the code would loop infinitely. At this point it is important to note that HARVESTER maintains the assignment to control variables fixed per run, i.e., it can only execute loops never (condition is `false`) or infinitely often (condition is `true`). In particular, in cases in which a loop condition does depend on an environment value, this may cause one of HARVESTER's dynamic runs to loop indefinitely. HARVESTER simply addresses this problem with a timeout on the overall execution time for every run of a slice. As evident from our experiments, this theoretical shortcoming does not pose a problem in practice. Developers intend computed values such as reflective call targets or telephone numbers for SMS scam to be independent of the execution environment.

In general, HARVESTER replaces only such conditionals that access values dependent on the execution environment. To be able to determine such conditionals, HARVESTER comes pre-customized with a configuration file listing fields and methods whose return values are known to depend on environment settings. Vidas et. al. [24] analyzed different techniques for Android emulator-detection and Maier et. al. [25] showed fingerprinting techniques for mobile sandboxes. We use the methods from these papers as a starting point for constructing the required lists. We believe the lists to be complete for current Android versions, but they can easily be extended.

The remainder of this section discusses the most important challenges that arise during backward slicing and how HARVESTER overcomes them.

Data Dependencies through Persistent Storage: Most applications use API classes such as `SharedPreferences` to persist data. Storage and retrieval can be distributed over the program. For instance, data can be stored into a file during application startup and read again after the reboot of the application—a common workflow also in current Android malware applications [26]. A slicing approach that does not model this data dependency between user actions would yield an incorrect slice that attempts to read non-existent data from an uninitialized data store. To handle these cases, HARVESTER resolves *all* calls within the analyzed bytecode 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 retained and all earlier values are overwritten. While a better handling might seem desirable, in our experiments the current solution proves sufficient to produce correct values for all logging points.

User Input: Further special handling is required for API calls that access environment values such as free-text user input. It is one major contribution of this work to show that within the slices that are frequently of interest to security analysts, such accesses to environment values, are, however, typically restricted to conditionals (see Section VI). Thus, they are removed by HARVESTER, as the respective expressions are replaced by Boolean control variables. Semantically, this restriction applies because obfuscators and malware authors seek to encode values independently of user input. The target of reflective call, for instance, is assumed to always be the same, regardless of the environment. In some few slices of interest, however, we found user input to be accessed also outside conditionals. In some cases this can simply happen because the slice is less precise than one would like it to be. To allow the execution also of such slices without user interaction, HARVESTER injects code to short-circuit the actual API calls that read out the UI, returning dummy values instead. Our experiments show this workaround, albeit somewhat crude, to be highly effective when applied to current malware.

Dynamic Code Loading and Native Code: Note that HARVESTER can also cope with dynamic code loading and native methods, as long as all logging points are contained within the APK's Dalvik bytecode. If, for instance, the value of an SMS message is computed by a native method, the slicer will declare this function as required and the dynamic execution will evaluate the function just as any other, invoking the same implementation that would also be invoked during normal app execution. Many current malware samples encode important values in native or dynamically loaded code, making this an essential feature [26].

Cut-Offs for Large Programs: For very large programs it may be infeasible to compute exact slices. 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 retained as they are, 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. To avoid uninitialized variables in this case, HARVESTER inserts artificial initialization statements that assign dummy values. As our experiments show, only few such dummy values are required in practice (see Section VI).

B. Dynamic Execution of Reduced APK

Part B in Figure 1 describes the *dynamic analysis* phase. HARVESTER assembles every slice computed during the static slicing phase within a single new method that becomes part of the reduced APK. The executor activity injected into the same APK file calls all these methods one after another, directly after the new app has been started, e.g., on an unmodified emulator or a stock Android phone. Since the slices are directly executed, 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 these interactions unnecessary. In fact, the reduced app does not contain any GUI elements from the original app at all. Figure 3 shows how the slice explained in Figure 2 would be executed. Executing this program will cause HARVESTER to report both possible valuations for `messageText`, along with the values for `clazz` and `method`.

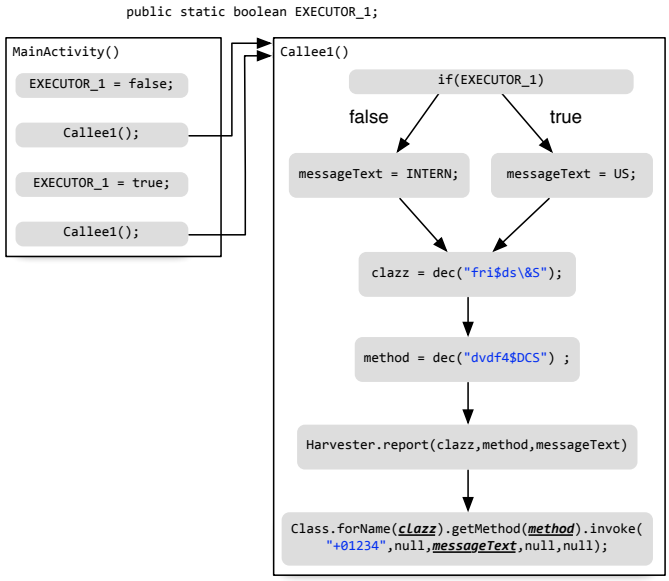

Fig. 3: Dynamic Execution of Reduced APK

As explained in Section V-A, slices are parametric and HARVESTER must explore every possible combination of branches to retrieve all values of interest at a given logging point. For the executor, this means that it must re-run the code slice for all possible combinations of these Boolean values. In the worst case (all conditions in the slice have to be replaced), 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. Only conditions inside the slice need to be considered. Thus, in practice, our experiments show n to be very limited ($n = 0.21$ per path on average over all our sample data, see Section VI). In the few cases in which it is not, our experiments show many of those paths to yield the same or at least very similar values. HARVESTER can thus be configured to sample only a predefined number of slice instances at random.

C. Runtime Value Injection

Part C in Figure 1 shows an optional step of HARVESTER, *runtime value injection*. This step can be useful to combine HARVESTER with existing off-the-shelf analysis tools, or to handle reflection. Static-analysis approaches require a call graph to determine potential targets for method invocations. For the large fraction of malware applications that are obfuscated using reflective method calls, such as the example in Listing 2, call graph construction fails. Some tools do not support reflective calls at all, while frameworks such as DOOP [27] implement a static best-effort solution but can still be fooled through string encoding. 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. To embed reflective calls into the program, HARVESTER uses the same approach originally taken in the TamiFlex tool [28]. Off-the-shelf analysis tools such as CHEX [29], SCanDroid [30] or FlowDroid [4] 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 for Android.

It is important to note that this very same mechanism is also what allows HARVESTER itself to extract runtime values from applications whose API calls have been obfuscated through reflection. In such cases, in phase A HARVESTER would first construct a partial call graph that is incomplete in the sense that it misses edges for reflective calls. It then extracts information about the parameters to those calls and inlines the calls as regular method calls. Finally, it reiterates the process, constructing a new, more complete call graph, and extracting further data values. This can be iterated up to a pre-defined number of times, or until a fixed point has been reached. This step is shown in Figure 1 by an edge from *Enhanced APK* to *Backward Slicer*.

VI. EVALUATION

We evaluated HARVESTER extensively on different sets of applications, one to address each of the following four research questions. In total, all sets together, comprise 16,799 apps. To the best of our knowledge, these sets faithfully model the state of the art in malware applications.

- **RQ1**: What is HARVESTER's precision and recall?
- **RQ2**: How does the recall of HARVESTER relate to that of existing static and dynamic-analysis approaches?
- **RQ3**: How efficient is HARVESTER?
- **RQ4**: Which interesting values does HARVESTER reveal?

In all experiments, the cut-offs were 20 for caller-slicing and 50 for callee-slicing which proved to be a reasonable tradeoff between recall and performance.

RQ1: What is HARVESTER*'s recall and precision?:* We evaluated HARVESTER's recall based on the coverage of logging points. Ideally, HARVESTER should cover every logging point. For the covered logging points we furthermore evaluated the precision and recall of the extracted runtime values. From our initial malware set of 16,799 samples, we took 12 different malware samples from 6 different malware families for an in-depth evaluation as shown in Table I. These samples were selected since they are representatives of various challenges for HARVESTER. Obad [1], for instance, is one of the most sophisticated malware families today. Many (FakeInstaller, GinMaster and Obad) are also highly obfuscated. These samples rely heavily on reflection to mask the targets of method calls. Another malware family, Pincer, is known to hinder dynamic analysis through anti-emulation techniques [9], [10]. Ssucl and Dougalek steal various private data items. We deliberately chose 12 complex samples only, since we sought to manually verify the precision and recall of HARVESTER.

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 that use constant values, which can be read off directly, and where consequently no backward-slicing and dynamic execution is necessary. Though the complexity of HARVESTER is not necessary to extract such constant values, HARVESTER discovers constant 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. The fraction directly next to the circle indicate the fraction of successfully extracted (non-constant) logging points, where the fraction in brackets show the fraction of successfully extracted logging points for constant values.

Table I shows two major facts: First, only 6.5% (bottom right corner $\frac{56}{860}$) of the extracted logging points contained a constant value. This confirms that a naive approach that only extracts constant values is not sufficient for our representative set of current malware. Furthermore, the table also shows that HARVESTER has a very high detection rate, since *green slices are bigger than the red slices* (bottom right corner).

In summary, the table shows that, averaged over all categories, HARVESTER detects at least one value for 86,6% (bottom right corner $\frac{745}{860}$) of all logging points. The fraction of missed logging points is due to HARVESTER's limitations (see Section IX) such as the lack of support for inter-component communication. 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 HARVESTER's limitations, which will be discussed in Section IX.

We then used those apps, for which at least one value of interest was discovered, to assess HARVESTER's precision and recall. Through manual inspection we were able to confirm that *all* values discovered by HARVESTER are actual runtime values, i.e., that HARVESTER has a precision of 100% on this data set. We furthermore evaluated the recall of the extracted SMS numbers, SMS messages and shell commands of our test data since those values are among the most important ones in a malware investigation. With the help of CodeInspect [31], an interactive bytecode debugger for Android applications, an independent ethical hacker manually reverse engineered and confirmed that HARVESTER extracted all runtime values for these categories. In other words, in those experiments also HARVESTER's recall is 100%.

HARVESTER was configured with a timeout of 10 minutes. This timeout caused the execution to abort in fewer than 1% of all cases. Dummy values due to cut-offs during the slicing (see Section V-A) only needed to be inserted in about 1% of all cases as well.

RQ2: How does the recall of HARVESTER *relate to existing static- and dynamic-analysis approaches?:* We next compare HARVESTER with purely static and purely dynamic approaches

	URI	Webview	SMS No.	SMS Text	File	Reflection	Shell Cmd	Sum
FakeInstaller (MD5)								
b702b545d521f129e8efc1631a3abcee	$\frac{3}{3}$ ($\frac{0}{3}$)	$\frac{4}{4}$ ($\frac{0}{4}$)			$\frac{6}{7}$ ($\frac{1}{7}$)	$\frac{6}{6}$ ($\frac{0}{6}$)		$\frac{19}{20}$ ($\frac{1}{20}$)
dd40531493f53456c3b22ed0bf3e20ef						$\frac{248}{280}$ ($\frac{0}{280}$)		$\frac{248}{280}$ ($\frac{0}{280}$)
GinMaster (MD5)								
0878b0bb41710324f7c0650daf6b0c93	$\frac{4}{12}$ ($\frac{4}{12}$)	$\frac{0}{2}$ ($\frac{1}{2}$)			$\frac{10}{14}$ ($\frac{2}{14}$)	$\frac{0}{3}$ ($\frac{1}{3}$)		$\frac{14}{31}$ ($\frac{8}{31}$)
ebe49b1b92a3b44eb159d15ca1f25c70	$\frac{7}{9}$ ($\frac{2}{9}$)	$\frac{1}{1}$ ($\frac{0}{1}$)			$\frac{25}{30}$ ($\frac{3}{30}$)			$\frac{33}{40}$ ($\frac{5}{40}$)
Obad (MD5)								
e1064bfd836e4c895b569b2de4700284						$\frac{185}{185}$ ($\frac{0}{185}$)		$\frac{185}{185}$ ($\frac{0}{185}$)
dd1a3ff43330165298db703f7f0626ce						$\frac{157}{161}$ ($\frac{2}{161}$)		$\frac{157}{161}$ ($\frac{2}{161}$)
Pincer (MD5)								
b2b7d5999dce0559d13ab06d30c2c6ec	$\frac{2}{2}$ ($\frac{0}{2}$)		$\frac{1}{2}$ ($\frac{1}{2}$)	$\frac{2}{2}$ ($\frac{0}{2}$)	$\frac{6}{13}$ ($\frac{6}{13}$)	$\frac{2}{3}$ ($\frac{1}{3}$)	$\frac{1}{1}$ ($\frac{0}{1}$)	$\frac{14}{23}$ ($\frac{8}{23}$)
9c9afd6b77d8d3a66a2db2d2cf0b94b3	$\frac{3}{3}$ ($\frac{0}{3}$)		$\frac{1}{2}$ ($\frac{1}{2}$)	$\frac{2}{2}$ ($\frac{0}{2}$)	$\frac{6}{13}$ ($\frac{6}{13}$)	$\frac{2}{3}$ ($\frac{1}{3}$)	$\frac{1}{1}$ ($\frac{0}{1}$)	$\frac{15}{24}$ ($\frac{8}{24}$)
Ssucl (MD5)								
f0bf007b3d2580297b208868425e98c7	$\frac{6}{9}$ ($\frac{2}{9}$)		$\frac{1}{1}$ ($\frac{0}{1}$)	$\frac{1}{1}$ ($\frac{0}{1}$)	$\frac{11}{22}$ ($\frac{8}{22}$)		$\frac{0}{2}$ ($\frac{2}{2}$)	$\frac{19}{35}$ ($\frac{12}{35}$)
c5a2d14bc52f109a06641c1f15e90985	$\frac{7}{10}$ ($\frac{2}{10}$)		$\frac{1}{1}$ ($\frac{0}{1}$)	$\frac{1}{1}$ ($\frac{0}{1}$)	$\frac{12}{19}$ ($\frac{4}{19}$)		$\frac{1}{3}$ ($\frac{2}{3}$)	$\frac{22}{34}$ ($\frac{8}{34}$)
Dougalek (MD5)								
95a04cfc5ed03c54d4749310ba29dda9	$\frac{2}{2}$ ($\frac{0}{2}$)		$\frac{2}{2}$ ($\frac{0}{2}$)	$\frac{2}{2}$ ($\frac{0}{2}$)	$\frac{10}{18}$ ($\frac{4}{18}$)			$\frac{16}{24}$ ($\frac{4}{24}$)
91d57eb7ee2582e0600f21b08dac9538	$\frac{3}{3}$ ($\frac{0}{3}$)							$\frac{3}{3}$ ($\frac{0}{3}$)
SUMMARY	$\frac{37}{53}$ ($\frac{10}{53}$)	$\frac{5}{7}$ ($\frac{1}{7}$)	$\frac{6}{8}$ ($\frac{2}{8}$)	$\frac{8}{8}$ ($\frac{0}{8}$)	$\frac{86}{136}$ ($\frac{34}{136}$)	$\frac{600}{641}$ ($\frac{5}{641}$)	$\frac{3}{7}$ ($\frac{4}{7}$)	$\frac{745}{860}$ ($\frac{56}{860}$)

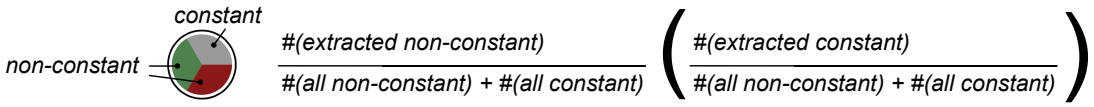

TABLE I: Recall-Evaluation of HARVESTER. Green slices: amount of logging points with non-constant values where a dynamic analysis is necessary for value extraction. Red slices: amount of missing logging points. Grey slices: amount of logging points with constant values where no static/dynamic analysis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

for automatically extracting values of interest from malicious applications.

Static Analysis: We compared HARVESTER with SAAF [17], 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 but uses traditional slicing. HARVESTER was evaluated on the same 6,100 malware samples as SAAF was evaluated (taken from MobileSandbox [32]). The logging points for both tools were the number and the corresponding message of text messages. The results for SAAF show that the tool has some issues with certain 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 cases, HARVESTER found more fragments of the target telephone number as SAAF.[2] In contrast, HARVESTER extracts the final, complete SMS numbers for all of the samples, even 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, opposed

[2]e.g. number `1065-5021-80133` in sample with MD5 hash `b238628ff1263c0cd3f0c03e7be53bfd`

to HARVESTER, SAAF cannot handle reflective calls with obfuscated target strings either. We further evaluated SAAF on current Android malware taken from Table I including the most sophisticated Android malware families: Obad, Pincer, Ssucl and Dougalek. SAAF was configured to extract values of interest for reflective method calls, SMS numbers and SMS messages. The tool was not able to extract any value of interest for Obad, Pincer and Ssucl. For Dougalek, SAAF found the same SMS numbers as HARVESTER, but was not able to extract SMS messages. The SMS numbers can be extracted in a static way (static backward slicing) since no obfuscation is applied to the constant string values. In summary, this shows that hybrid approaches such as HARVESTER can handle current malware samples more effectively than purely static ones like SAAF.

Dynamic Analysis: Extracting values of interest can also be achieved by executing the app and applying code coverage techniques [33]–[37] that try to reach the statement of the logging point. To evaluate HARVESTER on dynamic approaches, we randomly chose a set of 150 samples from 18 malware families from the Malware Genome Project [38]. We compared HARVESTER's recall with 5 different state-of-the-art testing-based approaches that were publicly available to us and could be setup with reasonable effort: Google's Monkey [33], PUMA [35], AndroidHooker [39], DynoDroid [34] and a naive approach that starts the app, waits for 10 seconds and quits the app. Unfortunately, we were not able to test Acteve [37] and SwiftHand [36] on our samples due to tool-internal issues.

The goal was to find the telephone numbers to which SMS messages are sent (all 150 samples contained at least one API call for sending SMS messages). To count how many logging points were reached by the dynamic testing tools, we instrumented the malware samples' bytecode to create a log entry directly before sending the message. After running the testing tools, we evaluated the log output taken from the Logcat tool. All tests were carried out on an Android 4.1 emulator (API version 16).

Table II shows that HARVESTER's recall is around four to six times higher than the one of current state-of-the-art dynamic analysis approaches. One reason for the particularly poor recall of the existing dynamic testing tools are emulator-detection techniques. These checks prevent the tools (which run the potentially malicious apps on an emulator for security reasons [32], [40]) from ever reaching a logging point in most malware samples.

Approaches	total logging-points covered
Simply open and close app	14.1%
Monkey	15.6%
PUMA	17.3%
AndroidHooker	16.2%
Dynodroid	22.3%
HARVESTER	83.4%

TABLE II: Measuring Recall of HARVESTER in Comparison to State-Of-The-Art Dynamic Testing Tools

As an example for such an emulator check, Listing 4 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 (line 5), the app sends out the premium SMS message (line 11). When a dynamic tool runs the app on an emulator, the contact database is usually empty and the logging point for sending SMS messages is thus never executed. 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 fixed target phone number can be retrieved by any tool. In such cases, HARVESTER reports a constant string with information about the source (e.g., contact database information). 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 only after the phone is rebooted. Apps from the *GPSSMSSpy* family act on incoming SMS messages. To obtain the respective runtime values, traditional dynamic approaches must 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.

To overcome simple environment checks, AndroidHooker [39] and Dynodroid [34] first prepare the emulator with fake "personal user data" such as contacts. Only afterwards, they install the application and exercise it. Both also send external events such as incoming SMS messages and AndroidHooker even reboots the emulator during the test to trigger actions that only happen at boot time. AndroidHooker 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. Due to such problems, AndroidHooker only covered 16.2% of all logging points. In only 10.67% of all apps it covered any logging point at all—a marginal improvement over running Monkey as is. In summary, these results show that current state-of-the art testing tools are not sufficient for revealing malicious behavior of current state-of-the art malicious applications. HARVESTER succeeds in all cases, 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 executed directly and unconditionally.

All in all, dynamic tools only reach a small fraction of all logging points for these malware samples. It is worth mentioning that a naive approach that starts an app, waits for ten seconds and closes the app, produces similar results

```
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 (contacts.getCount() > 0) {
6      do {
7        int colIdx = contacts.getColumnIndex("data1");
8        String telNo = contacts.getString(colIdx);
9        sms.sendTextMessage(telNo, null, "I take
       pleasure in hurting small animals, just thought you
       should know that", ...);
10     } while (contacts.moveToNext());
11     sms.sendTextMessage("73822", null, "text", ...);
12   }
13 }
```

Listing 4: "DogWars" Game from Malware Genome Project

(first line in table) as Google's Monkey approach. HARVESTER, on the other hand, covers 83.4% of all logging points and thus shows a much higher recall.

RQ3: How efficient is HARVESTER*?:* App Stores such as the Google Play Store receive thousands of new or updated Android apps per day [41] which they need to check for malicious behavior. Therefore, one requires fast tools which scale to the size of the market. We tested HARVESTER on our full set of 16,799 malware samples (which includes all samples from the previous sections). We configured HARVESTER with two logging-points (SMS phone numbers and the respective text messages) for each sending SMS API call included in the app's bytecode. We focused on SMS numbers and messages since SMS trojans are among the most sophisticated malware apps 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 evaluation reported in this section was run on a compute server with 64 Intel Xeon E5-4650 cores running Ubuntu Linux 14.04 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 about 2.5 minutes per application. This shows that HARVESTER can be used for mass analyses and delivers results very quickly. On average over all slices in all our samples, HARVESTER had to try different values for 0.21 `EXECUTOR` flags per slice. The highest average number of `EXECUTOR` flags we encountered per slice in a single app was 1.31.

RQ4: Which interesting values does HARVESTER *reveal?:* We next report interesting values that HARVESTER extracted from malware applications. Our analysis is based on our full sample set of 16,799 malware apps. 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 reveal all of these findings. HARVESTER found a lot of cases where malicious applications used reflective method calls to hide sensitive method calls such as "getDeviceId" or "sendTextMessage". In some applications even the reflective calls themselves were again called via reflection to produce a multi-stage obfuscation. HARVESTER is able to extract the called methods in all of our samples.

HARVESTER also discovered that current SMS trojans are far more sophisticated then just sending a hard-coded number of premium-rate SMS messages per time frame or upon certain actions (e.g., every time the victim opens the malicious application). Some SMS trojans store the number of messages sent in `SharedPreferences`, a key-value storage provided by the Android framework. HARVESTER uncovers 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 messages. Since the same commands reappear in many samples, they also could be used for blacklisting.

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 [13]. 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 can fully automatically extract the WhatsApp encryption key by obtaining the values passed to the constructor of the `SecretKeySpec` class.

A more detailed overview of HARVESTER's findings can be found in our technical report [42].

VII. CASE-STUDIES

While the previous section focused on how well HARVESTER can extract runtime values from (obfuscated) Android applications, we report on two case studies that assess how existing off-the-shelf static and dynamic analysis tools can benefit from a lightweight integration with HARVESTER. Figure 4 shows which step of our approach can be further used for the improvement of static and dynamic analysis tools.

Improvements to Static Analysis Tools

For the first case study, we used HARVESTER to inject information about discovered reflective calls into the original app's bytecode (phase C in Figure 4). We then compared the recall of the FlowDroid [4] static data flow tracker on real-world malware applications with and without this call information. For this comparison, we chose the Fakeinstaller.AH [2] malware family [3] 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.

To evaluate in more detail how HARVESTER improves the precision and recall of existing tools on obfuscated applications, we tested FlowDroid on ten randomly-picked applications from DroidBench [4] which we obfuscated using DexGuard [16]. All API method calls were replaced with reflective calls on encrypted strings. Table III compares the detection rate of FlowDroid on the obfuscated applications without applying HARVESTER (*BEFORE* - column 2 and 4) to the respective detection rates after applying 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 V-C), FlowDroid found the same leaks as in the unobfuscated original version. The enhanced APK with the injected runtime-values is shown in Figure 4. In "PrivacyDataLeak3", FlowDroid always misses one of the two leaks, even in the original, unobfuscated file, for reasons unrelated to the work presented here.

[3] Sample MD5: 38a9ed0b5577af6392096b4dc4a61e02

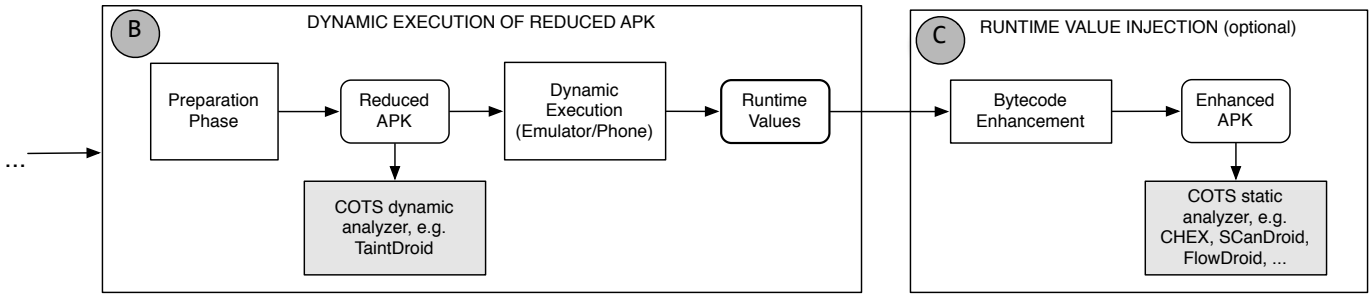

Fig. 4: Workflow for Improving Static and Dynamic Taint Flow Analyses

Improvements to Dynamic Analysis Tools

Dynamic analysis tools can only inspect code that is actually executed. If an analyst wants to find malicious behavior in a suspicious app using such a dynamic tool, she must therefore ensure that the malicious code is indeed triggered. As we have shown in our evaluation in Section VI, current testing approaches for Android, however, often fail to trigger the malicious behavior in current malware samples.

HARVESTER's static slicer extracts exactly the code required for computing a specific value of interest. Afterwards, only this code is run on an emulator or a real phone. Most importantly, the reduced code executed by HARVESTER does not include any emulator checks or other techniques targeted at hindering dynamic analysis. Furthermore, no user interaction with the application is required anymore, eliminating code coverage issues with existing input generation approaches. Running existing off-the-shelf dynamic analysis tools not on the original APK, but on the reduced APK (see phase B in Figure 4) created by HARVESTER can thus greatly improve their recall as we show in this section. In our second case study, we compare the recall of the well-known dynamic taint tracker TaintDroid on the original APK file and on HARVESTER's reduced version.

In an approach similar to Anubis [43], TaintDroid 4.1 was run inside the emulator on the Tapsnake [38] malware sample[4] which steals location data only after a delay of 15 minutes [44]. On the original malware, the analyst needs to know that she has to wait this time. With the app reduced by HARVESTER's slicing approach, TaintDroid reports the leak instantly, without any UI interaction.

We again took 10 randomly-picked examples from DroidBench and obfuscated them with DexGuard. Table III compares the recall of TaintDroid on the obfuscated apps with the recall after using HARVESTER's value injection. In the original app, TaintDroid missed leaks depending on user actions such as in "Button3". On apps containing emulator-detection checks it failed as well. When running the slices extracted by HARVESTER (see "Reduced APK" in Figure 4), 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.

[4]Sample MD5: 7937c1ab615de0e71632fe9d59a259cf

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

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

TABLE III: 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.

VIII. FURTHER USES CASES

The primary goal of HARVESTER is to extract runtime values, even from obfuscated Android applications. Aside from improving the effectiveness of static and dynamic taint analyses as shown in Section VII we now discuss further uses cases that we plan to explore in future work, at the same time inviting other researchers to join us in this process.

Simplifying inter-component communication

In Android, inter-application and inter-component communication is usually performed using *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. Therefore, current state-of-the-art tools such as EPICC [6], IccTA [5] and IC3 [45] can only conservatively over-approximate in such cases, which leads to potential false positives. With HARVESTER, the actual runtime values can be integrated into the app as constant strings, reducing the risk of such false positives.

Improving Sandbox Output

Different sandboxing approaches such as Andrubis [40] or Mobile Sandbox [32] apply different static as well as dynamic analysis techniques for producing a security-report of an application. Most of the time, these approaches apply lightweight code-analysis techniques, such as finding *statically coded URLs with the help of a regular expression* [32]. However, this results in little to no output for obfuscated applications that try to hide their URL, for instance. HARVESTER can help recover these values as part of the toolchain.

Improving Malware Detection Approaches

There exist different machine learning approaches [46]–[49] that try to 'learn' how a benign or malicious application looks like in order to find new malicious applications. These approaches are trained with different features on a set of applications. However, if the feature set is not significant enough to differentiate between malicious and benign applications, it produces too many false positives. HARVESTER's output can improve this situation by defining precise features that couldn't be used with previous approaches. Example features would be *runtime values passed into method calls* or *resolved reflective method calls* (e.g. obfuscated sensitive API calls).

Improving Fuzzing Approaches

Fuzzing approaches are an essential technique for a fast detection of critical security vulnerabilities in various applications [50]. Fuzzing, for instance, helped identify the critical Stagefright Vulnerability [51] in the Android OS. However, most of the fuzzing approaches rely on an input set. This is especially problematic if one needs a specific input format, such as incoming SMS messages from a C&C server, or a specific intent, in order to trigger a certain security vulnerability. HARVESTER can be used to generate such proper input sets for fuzzing tools based on concrete runtime values.

IX. LIMITATIONS AND SECURITY ANALYSIS

While HARVESTER improves over the state of the art significantly, like any approach it comes with some limitations. We next discuss those limitations and how malware authors could potentially exploit them. Attempting to overcome those limitations will make for an interesting piece of future research.

Attacking Timeout Mechanism

To compute the values of interest, HARVESTER executes the extracted slices. Execution ends if either all values of interest have been computed, or a timeout occurs. An attacker can theoretically exploit this timeout by deliberately creating large apps with many data-flow dependencies on the values of interest. Such an attack would lead to larger slices, and hence, longer execution times per slice, making timeouts - and thus missed values - more likely. An analyst can, however, easily increase the timeouts if she detects that they happen too frequently and results are poor. Additionally, one has to keep in mind that such *Data- and Control-flow obfuscations* also increase the code size and execution time of the original app. This would severely limit the practical applicability of such obfuscators.

Overwhelming the Analyst with Spurious Values

Since HARVESTER over-approximates the paths to be executed, it may yield false positives, i.e., values that cannot be computed by the original program in any given environment. The code in Listing 5 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, reporting the spurious value `0000`. For future work, we will additionally add semantic checks that try to verify the validity of an environment-check (e.g., whether `if(simCountryIso().equals("XX"))` is a valid check or not) to eliminate fake environment checks.

```
1 String number = null;
2 if(simCountryIso().equals("DE"))
3   number = 9371;
4 if(simCountryIso().equals("XX"))
5   number = 0000;
6 sendTextMessage(number, "msg");
```

Listing 5: Path Over-Approximation

Hiding Logging Points

HARVESTER is currently implemented for the Dalvik part of Android applications. Section V-A described that HARVESTER is able to handle applications containing native method calls as long as the logging point is still contained in the Dalvik code. If, for instance, an SMS message is sent by native code, this hidden call to `sendTextMessage()` cannot be used as a logging point. If an attacker, hides the complete computation of the value of interest in native code and never yields the computed result back to the Dalvik layer, HARVESTER will not be able to extract these values. However, according to previous research, current state-of-the art banking trojans [26] use native code mainly to hide sensitive information but leak the data in the Dalvik part. In such cases, HARVESTER can extract this sensitive information, returned by the native methods, without problems.

HARVESTER can succeed, however, if the app loads Dalvik code dynamically. In such a situation, the analyst would first run HARVESTER once to obtain the dynamically loaded code (which is just another runtime value), and then once again to extract the values of interest. In the first run, the dynamically loaded code will be merged into the dex-file and in the second step the hidden logging point in the merged dex-file will be recognized and analyzed by HARVESTER.

Attacking Static Backward Slicing

Attackers could also focus on the static backward slicing. To compute a static program slice, a complete callgraph is indispensable, as with an incomplete call graph the slices may be incomplete as well. If an app therefore contains multiple layers of reflective calls, the slices computed by HARVESTER will be incomplete. However, since HARVESTER is able to replace reflective method calls with their original call targets (see Section V-C), an analyst can run HARVESTER multiple times, removing one layer of reflective calls per run. In the end, HARVESTER is able to construct a complete callgraph and, hence, a complete slice. The same technique of multiple executions can also be used if reflective calls occur in the code that computes the target of further reflective calls.

At the moment, HARVESTER does not support slices that span multiple Android components. If a value, for instance, is computed in one activity and then sent to a second one which then contains the logging point, this value will be missed. In the future, we plan to extend HARVESTER with support for inter-component communication, by integrating an existing inter-component analysis tool such as EPICC [6] or IC3 [45]. Since both tools are based on Soot, just like HARVESTER, they should be directly compatible.

Attacking Data Dependency

We assume the values of interest not to be data dependent on environment values. For current malware this proves to be a reasonable assumption. If malware developers were to introduce such dependencies in the future, one could react by extending HARVESTER to detect and report such cases to a human analyst. This can be achieved with the help of a static data flow tracking approach that tries to identify whether the logging point is data dependent on an environment value. While this approach can be attacked due to its static nature, such a detection would significantly raise the bar for an attacker. Note that HARVESTER can be applied iteratively to remove layers of obfuscation (e.g., replace reflective calls with direct method invocations). In every run, the app gets simpler and, thus, more accessible to such static detections.

Attacking the Completeness of Values of Interest

If values of interest are computed using data from external resources such as servers on the web, we assume this data to be static. If, for instance, a remote server returns different target phone numbers for an SMS scam every day, HARVESTER will only be able to recover the value of interest for the present day.

X. 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 ded [52] or Dare [53] 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 [4] or DroidSafe [54] are static taint analysis tools which determine whether sensitive information is leaked in an Android application. Due to their static nature, they cannot handle reflective calls whose target class or method name is decrypted or concatenated dynamically at runtime. CHEX [29], IC3 [45] or Amandroid [55] are static approaches that face the problem of inter-component data flow tracking in Android applications. Just like FlowDroid, the approaches rely on a complete call graph and thus fail if call targets are obfuscated using reflection. They would thus also benefit from our runtime value injection for a more complete analysis. SAAF [17] 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 [56] 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 [14] is a dynamic data-flow tracker which detects leaks of sensitive information at runtime. Other techniques such as Aurasium [57] 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. Dynamic determinacy analysis [58] is an approach for identifying values that always have the same value in all executions of a program, regardless of the input values. This model, however, does not allow for sets of values that are constant for a given environment only.

Hybrid Analysis: TamiFlex [28] 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 does not support Android, however, and employs no slicing. Instead, it always executes a full, single run, leaving open how full coverage of callees is to be achieved. AppDoctor [59] 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 [60] 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 [61] 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. Rozzle [62], a tool for de-cloaking internet malware has a similar goal as HARVESTER, but has its limitation in triggering the malicious behavior. For instance, it can not handle timing or logic bombs. Zhou et. al. [63] present an approach that is, just like HARVESTER, based on slicing and execution. They, however, execute the app inside a custom interpreter which is also responsible for steering the execution into specific branches. As this approach completely replaces the Android OS, it requires a very precise model of the OS and its libraries. Roundy et al. [64] combine static and dynamic code analysis in order to make the CFG more precise in cases where malware is packed, obfuscated or dynamically loads additional code. Zhao et al. [65] provide an approach for extracting runtime values for native binaries. They also combine static backward slicing with dynamic code execution, but their extracted slice contains an unmodified code, including conditions. This results in a lack of extracting values of interest since only one path will be executed during runtime.

UI-Automation: SwiftHand [36] 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. On complex user interfaces, however, SwiftHand's code coverage can fall under 40% according to the numbers stated in the paper. Code that is only executed in specific environments (e.g., depending on data loaded from the Internet) might not be reached at all. Dynodroid [34] instruments the Android framework for capturing events from unmodified applications, generated both by automatic techniques such as MonkeyRunner [33] and by human analysts. On average, it achieves a code coverage of 55%. Brahmastra [66] is another UI-testing tool that combines static analysis with bytecode rewriting in order to directly execute certain code statements. Since the tool relies on a complete static callgraph, it has its limitation in applications that are obfuscated with reflective method calls such as the one used in the Obad malware family. AppsPlayground [67] uses an enhanced version of TaintDroid [14] 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 use random testing guided by heuristics leading to a code coverage of about 33%. As HARVESTER directly executes the code fragments of interest, it does not need a method for UI automation, avoiding the problem of poor coverage and recall.

XI. CONCLUSIONS

In this paper, we 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 other things, plain-text telephone numbers of SMS trojans, command and control messages of bots, and reflective call targets of various types of malware. Opposed to current state-of-the art UI automation approaches HARVESTER yields an almost perfect coverage of logging points. 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 significantly improves the recall of current static and dynamic data-flow analysis tools. On average, HARVESTER analyzes an application in less than three minutes, yielding many dynamically computed runtime values that no previous automated approach was able to retrieve.

Acknowledgements: This work was supported by the BMBF within EC SPRIDE and ZertApps, by the Hessian LOEWE excellence initiative within CASED, and by the DFG through the projects TESTIFY and RUNSECURE, the Collaborative Research Center CROSSING and the Priority Program 1496 Reliably Secure Software Systems. We would like to thank our shepherd Christopher Kruegel and all anonymous reviewers throughout the project for improving the paper and HARVESTER.

REFERENCES

[1] E. Tinaztepe, D. Kurt, and A. Güleç, "Android obad," COMODO, Tech. Rep., Jul. 2013.

[2] F. Ruiz, "Fakeinstaller leads the attack on android phones," McAfee Labs Website, Oct 2012, https://blogs.mcafee.com/mcafee-labs/fakeinstaller-leads-the-attack-on-android-phones.

[3] F-Secure Labs, "Trojan:android/pincer.a," Blog, Apr. 2013, https://www.f-secure.com/weblog/archives/00002538.html.

[4] C. Fritz, S. Arzt, S. Rasthofer, 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 *PLDI '14*, Jun. 2014. [Online]. Available: http://www.bodden.de/pubs/far+14flowdroid.pdf

[5] L. Li, A. Bartel, T. F. Bissyande, J. Klein, Y. L. Traon, S. Arzt, S. Rasthofer, E. Bodden, D. Octeau, and P. McDaniel, "IccTA: Detecting inter-component privacy leaks in android apps," in *ICSE '15*, 2015.

[6] D. Octeau, P. McDaniel, S. Jha, A. Bartel, E. Bodden, J. Klein, and Y. Le Traon, "Effective inter-component communication mapping in android with epicc: An essential step towards holistic security analysis," in *USENIX Security '13*. Berkeley, CA, USA: USENIX Association, 2013, pp. 543–558.

[7] D. Octeau, D. Luchaup, M. Dering, S. Jha, and P. McDaniel, "Composite constant propagation: Application to android inter-component communication analysis," in *Proceedings of the 37th International Conference on Software Engineering (ICSE)*, 2015.

[8] K. Coogan, S. Debray, T. Kaochar, and G. Townsend, "Automatic static unpacking of malware binaries," in *WCRE '09*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 167–176.

[9] T. Petsas, G. Voyatzis, E. Athanasopoulos, M. Polychronakis, and S. Ioannidis, "Rage against the virtual machine: Hindering dynamic analysis of android malware," in *EuroSec '14*. New York, NY, USA: ACM, 2014, pp. 5:1–5:6.

[10] T. Vidas and N. Christin, "Evading Android runtime analysis via sandbox detection," in *ASIACCS '14*, Kyoto, Japan, Jun. 2014.

[11] L. Kelion, "Android adware 'infects millions' of phones and tablets," BBC, Feb. 2015, http://www.bbc.com/news/technology-31129797.

[12] C. S. Jensen, M. R. Prasad, and A. Møller, "Automated testing with targeted event sequence generation," in *ISSTA '13*. New York, NY, USA: ACM, 2013, pp. 67–77.

[13] Google Play, "Whatsapp messenger," Google PlayStore Website, Mai 2014, https://play.google.com/store/apps/details?id=com.whatsapp.

[14] W. Enck, P. Gilbert, B. gon Chun, L. P. Cox, J. Jung, P. McDaniel, and A. Sheth, "Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones," in *OSDI '10*, 2010, pp. 393–407.

[15] F-Secure, "Mobile threat report q1 2014," Apr. 2014, http://www.f-secure.com/static/doc/labs_global/Research/Mobile_Threat_Report_Q1_2014_print.pdf.

[16] S. A. C. Technology, "Dexguard," Saikoa Website, Feb 2014, http://www.saikoa.com/dexguard.

[17] J. Hoffmann, M. Ussath, T. Holz, and M. Spreitzenbarth, "Slicing droids: Program slicing for smali code," in *SAC '13*. New York, NY, USA: ACM, 2013, pp. 1844–1851. [Online]. Available: http://doi.acm.org/10.1145/2480362.2480706

[18] T. Raffetseder, C. Kruegel, and E. Kirda, "Detecting system emulators," in *ISC '07*. Berlin, Heidelberg: Springer-Verlag, 2007, pp. 1–18.

[19] S. Rasthofer, S. Arzt, and E. Bodden, "A machine-learning approach for classifying and categorizing android sources and sinks," *NDSS '14*, February 2014.

[20] Z. Xu, J. Zhang, G. Gu, and Z. Lin, "Goldeneye: Efficiently and effectively unveiling malwares targeted environment," in *Research in Attacks, Intrusions and Defenses*, ser. Lecture Notes in Computer Science, A. Stavrou, H. Bos, and G. Portokalidis, Eds. Springer International Publishing, 2014, vol. 8688, pp. 22–45.

[21] A. Moser, C. Kruegel, and E. Kirda, "Exploring multiple execution paths for malware analysis," in *Proceedings of the 2007 IEEE Symposium on Security and Privacy*, ser. SP '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 231–245.

[22] M. Weiser, "Program slicing," in *ICSE '81*. Piscataway, NJ, USA: IEEE Press, 1981, pp. 439–449.

[23] F. Tip, "A survey of program slicing techniques." Amsterdam, The Netherlands, The Netherlands, Tech. Rep., 1994.

[24] T. Vidas and N. Christin, "Evading android runtime analysis via sandbox detection," in *ASIA CCS '14*. New York, NY, USA: ACM, 2014, pp. 447–458.

[25] D. Maier, T. Muller, and M. Protsenko, "Divide-and-conquer: Why android malware cannot be stopped," in *ARES '14*, Sept 2014, pp. 30–39.

[26] S. Rasthofer, I. Asrar, S. Huber, and E. Bodden, "How current android malware seeks to evade automated code analysis," in *9th International Conference on Information Security Theory and Practice (WISTP'2015)*.

[27] M. Bravenboer and Y. Smaragdakis, "Strictly declarative specification of sophisticated points-to analyses," in *OOPSLA '09*, 2009, pp. 243–262.

[28] E. Bodden, A. Sewe, J. Sinschek, M. Mezini, and H. Oueslati, "Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders," in *ICSE '11*. New York, NY, USA: ACM, 2011, pp. 241–250.

[29] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang, "Chex: statically vetting android apps for component hijacking vulnerabilities," in *CCS '12*, 2012, pp. 229–240.

[30] P. Adam, A. Chaudhuri, and J. Foster, "Scandroid: Automated security certification of android applications," in *SP '09*, 2009.

[31] Secure Software Engineering Group Darmstadt, "Codeinspect binary android analysis," Blog, http://sseblog.ec-spride.de/tools/codeinspect/.

[32] M. Spreitzenbarth, F. Freiling, F. Echtler, T. Schreck, and J. Hoffmann, "Mobile-sandbox: Having a deeper look into android applications," in *SAC '13*. New York, NY, USA: ACM, 2013, pp. 1808–1815.

[33] Google Developers, "monkeyrunner," Google Developer Website, Mai 2014, http://developer.android.com/tools/help/monkeyrunner_concepts.html.

[34] A. Machiry, R. Tahiliani, and M. Naik, "Dynodroid: An input generation system for android apps," in *ESEC/FSE '13*. New York, NY, USA: ACM, 2013, pp. 224–234.

[35] S. Hao, B. Liu, S. Nath, W. G. Halfond, and R. Govindan, "Puma: Programmable ui-automation for large-scale dynamic analysis of mobile apps," in *Proceedings of the 12th Annual International Conference on Mobile Systems, Applications, and Services*, ser. MobiSys '14. New York, NY, USA: ACM, 2014, pp. 204–217.

[36] W. Choi, G. Necula, and K. Sen, "Guided gui testing of android apps with minimal restart and approximate learning," in *OOPSLA '13*. New York, NY, USA: ACM, 2013, pp. 623–640.

[37] S. Anand, M. Naik, M. J. Harrold, and H. Yang, "Automated concolic testing of smartphone apps," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, ser. FSE '12. New York, NY, USA: ACM, 2012, pp. 59:1–59:11.

[38] Y. Zhou and X. Jiang, "Dissecting android malware: Characterization and evolution," in *SP '12*. Washington, DC, USA: IEEE Computer Society, 2012, pp. 95–109.

[39] G. Bossert and D. Kirchner, "How to play hooker: Une solution d'analyse automatisée de markets android."

[40] M. Lindorfer, M. Neugschwandtner, L. Weichselbaum, Y. Fratantonio, V. van der Veen, and C. Platzer, "Andrubis - 1,000,000 Apps Later: A View on Current Android Malware Behaviors," in *Proceedings of the the 3rd International Workshop on Building Analysis Datasets and Gathering Experience Returns for Security (BADGERS)*, 2014.

[41] A. Stats, "Number of android applications," Android Statistics Page of AppBrain, March 2014, http://www.appbrain.com/stats/number-of-android-apps.

[42] S. Rasthofer, S. Arzt, M. Miltenberger, and E. Bodden, "Harvesting runtime data in android applications for identifying malware and enhancing code analysis," EC SPRIDE, Tech. Rep. TUD-CS-2015-0031, Feb. 2015.

[43] I. S. S. Lab, "Anubis - malware analysis for unknown binaries," Anubis Website, mai 2014, http://anubis.iseclab.org.

[44] Z. Yang, M. Yang, Y. Zhang, G. Gu, P. Ning, and X. S. Wang, "Appintent: analyzing sensitive data transmission in android for privacy leakage detection," in *CCS '13*. New York, NY, USA: ACM, 2013, pp. 1043–1054.

[45] D. Octeau, D. Luchaup, M. Dering, S. Jha, and P. McDaniel, "Composite Constant Propagation: Application to Android Inter-Component Communication Analysis," in *Proceedings of the 37th International Conference on Software Engineering (ICSE)*, May 2015.

[46] S. Chakradeo, B. Reaves, P. Traynor, and W. Enck, "Mast: Triage for market-scale mobile malware analysis," in *WiSec '13*. New York, NY, USA: ACM, 2013, pp. 13–24.

[47] A. Gorla, I. Tavecchia, F. Gross, and A. Zeller, "Checking app behavior against app descriptions," in *ICSE '14*, May 2014.

[48] D. Arp, M. Spreitzenbarth, M. Hübner, H. Gascon, K. Rieck, and C. Siemens, "Drebin: Effective and explainable detection of android malware in your pocket," in *NDSS'14*, San Diego, CA, February 2014.

[49] V. Avdiienko, K. Kuznetsov, A. Gorla, A. Zeller, S. Arzt, S. Rasthofer, and E. Bodden, "Mining apps for abnormal usage of sensitive data," in *Software Engineering (ICSE), 2015 IEEE/ACM 37th IEEE International Conference on*, vol. 1, May 2015, pp. 426–436.

[50] P. Godefroid, M. Y. Levin, and D. Molnar, "Sage: Whitebox fuzzing for security testing," *Queue*, vol. 10, no. 1, pp. 20:20–20:27, Jan. 2012.

[51] J. Drake, "Stagefright: Scary code in the heart of android," BlackHat USA 2015, Aug. 2015.

[52] W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri, "A study of android application security," in *USENIX Security '11*. Berkeley, CA, USA: USENIX Association, 2011, pp. 21–21.

[53] D. Octeau, S. Jha, and P. McDaniel, "Retargeting android applications to java bytecode," in *FSE '12*. New York, NY, USA: ACM, 2012, pp. 6:1–6:11.

[54] M. I. Gordon, D. Kim, J. Perkins, L. Gilham, N. Nguyen, and M. Rinard, "Information-flow analysis of Android applications in DroidSafe," in *Proceedings of the 22nd Annual Network and Distributed System Security Symposium (NDSS'15)*, 2015.

[55] F. Wei, S. Roy, X. Ou, and Robby, "Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '14. New York, NY, USA: ACM, 2014, pp. 1329–1341.

[56] J. Oberheide and C. Miller, "Dissecting the android bouncer," *SummerCon2012, New York*, 2012.

[57] R. Xu, H. Saïdi, and R. Anderson, "Aurasium: practical policy enforcement for android applications," in *USENIX Security '12*, 2012.

[58] M. Schäfer, M. Sridharan, J. Dolby, and F. Tip, "Dynamic determinacy analysis," in *PLDI '13*. New York, NY, USA: ACM, 2013, pp. 165–174.

[59] G. Hu, X. Yuan, Y. Tang, and J. Yang, "Efficiently, effectively detecting mobile app bugs with appdoctor," *EuroSys*, 2014.

[60] M. Zhang and H. Yin, "Appsealer: Automatic generation of vulnerability-specific patches for preventing component hijacking attacks in android applications," in *NDSS'14*, San Diego, CA, Feb. 2014.

[61] D. Sounthiraraj, J. Sahs, G. Greenwood, Z. Lin, and L. Khan, "Smvhunter: Large scale, automated detection of ssl/tls man-in-the-middle vulnerabilities in android apps," in *NDSS'14*, San Diego, CA, February 2014.

[62] C. Kolbitsch, B. Livshits, B. Zorn, and C. Seifert, "Rozzle: De-cloaking internet malware," in *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, ser. SP '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 443–457.

[63] Y. Zhou, L. Wu, Z. Wang, and X. Jiang, "Harvesting developer credentials in android apps," in *Proceedings of the 8th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, ser. WiSec '15, 2015.

[64] K. A. Roundy and B. P. Miller, "Hybrid analysis and control of malware," in *Proceedings of the 13th International Conference on Recent Advances in Intrusion Detection*, ser. RAID'10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 317–338.

[65] Z. Zhao, G.-J. Ahn, and H. Hu, "Automatic Extraction of Secrets from Malware," in *Proceedings of the 18th Working Conference on Reverse Engineering*, M. Pinzger, D. Poshyvanyk, and J. Buckley, Eds. IEEE Computer Society, 2011, pp. 159–168.

[66] R. Bhoraskar, S. Han, J. Jeon, T. Azim, S. Chen, J. Jung, S. Nath, R. Wang, and D. Wetherall, "Brahmastra: Driving apps to test the security of third-party components," in *Proceedings of 23rd USENIX*. Berkeley, CA, USA: USENIX Association, 2014, pp. 1021–1036.

[67] V. Rastogi, Y. Chen, and W. Enck, "Appsplayground: Automatic security analysis of smartphone applications," in *CODASPY '13*. New York, NY, USA: ACM, 2013, pp. 209–220.