Exploring Syscall-Based Semantics Reconstruction of Android Applications

Dario Nisi[1], Antonio Bianchi[2], and Yanick Fratantonio[1]

[1]EURECOM
[2]Purdue University
dario.nisi@eurecom.fr, antoniob@purdue.edu, yanick.fratantonio@eurecom.fr

Abstract

Within the realm of program analysis, dynamic analysis approaches are at the foundation of many frameworks. In the context of Android security, the vast majority of existing frameworks perform API-level tracing (i.e., they aim at obtaining the trace of the APIs invoked by a given app), and use this information to determine whether the app under analysis contains unwanted or malicious functionality. However, previous works have shown that these API-level tracing and instrumentation mechanisms can be easily evaded, regardless of their specific implementation details. An alternative to API-level tracing is syscall-level tracing. This approach works at a lower level and it extracts the sequence of syscalls invoked by a given app: the advantage is that this approach can be implemented in kernel space and, thus, it cannot be evaded and it can be very challenging, if not outright impossible, to be detected by code running in user space. However, while this approach offers more security guarantees, it is affected by a significant limitation: most of the semantics of the app's behavior is lost. These syscalls are in fact low-level and do not carry as much information as the highly semantics-rich Android APIs. In other words, there is a significant *semantic gap*.

This paper presents the first exploration of how much it would take to bridge this gap and how challenging this endeavor would be. We propose an approach, an analysis framework, and a pipeline to gain insights into the peculiarities of this problem and we show that it is much more challenging than what previously thought.

1 Introduction

In the realm of malware analysis for Android apps, dynamic analysis approaches and instrumentation techniques are at the foundation of virtually all existing analysis frameworks, developed by both academia and industry [2, 6, 10, 14, 17–19, 21, 31, 34]. While dynamic analysis approaches can take many forms, they all share one key aspect: given an application, the goal is to "capture" all actions it performs during its execution. To this end, these apps are run in an instrumented environment, which records a trace of the app's behavior.

API-level tracing. In Android, most of these approaches aim at producing a list of high-level API calls performed by the app under analysis. These high-level API calls are Java methods exposed by the Android framework, a vast extension of the Java SDK. These methods include standard Java methods (e.g., string operations, networking primitives), as well as a large corpus of Android-specific methods, such as APIs dealing with building Android user interfaces, inter-app interactions, reading values from device's sensors, sending and receiving text messages. Having access to an accurate trace of invoked APIs is of great importance. In fact, these APIs capture the high-level, *semantic-rich* behavior of an app and allow both human analysts and automated approaches to detect and characterize both malicious and unsafe actions the app could perform.

These approaches work by heavily instrumenting the app itself or the execution environment (e.g., by function hooking). Unfortunately, to date, *all* current API-level instrumentations can be easily detected and bypassed [8]. The key problem is that these instrumentation mechanisms all introduce visible instrumentation artifacts, which co-exist within the same security boundary as the app itself. Acquiring these high-level traces require heavy instrumentation, which is hard to implement efficiently and it is trivially detectable by malware, which could decide not to show any malice when instrumentation is detected [33]. Android apps can also contain components written in native code, whose behavior cannot be captured if the app's instrumentation is only performed at the Java API level. More importantly, previous works have shown that the mere presence of native code can make the results of the analysis of the Java layer not only detectable and evadable, but even misleading [8]. In fact, since native code components and Java code run within the same security boundary, native components could surreptitiously modify the intended functionality of Java components making high-level recordings of an app's behavior completely unreliable. These issues severely affect the reliability of acquiring high-level API-based traces.

Syscall-level tracing. A different approach consists in capturing the actions performed by an app by recording its low-level interactions with the operating system, specifically, by recording the system calls (syscalls) it invokes [11,30]. This approach is not affected by the limitations mentioned above. In fact, regardless of the language used to implement the different app's components, to interact with the operating system, the app *needs* to eventually invoke a system call. Moreover, this kind of instrumentation is harder to detect, since it can be easily implemented entirely by code running in kernel mode, not visible to the analyzed malicious app.

Bridging the semantic gap. Given the security guarantees that approaches based on syscall-level analysis would get us, it is clear that, ideally, this approach should be preferred. In practice, however, the information they extract is too low level, making their results difficult to be interpreted. The conceptual problem is that, in the general case, it is challenging to recover the high-level *semantics* of an app's behavior solely starting from the list of recorded syscalls. For instance, even a simple operation, such as instantiating an SSL connection to a remote server, which an Android app can perform by invoking a single high-level API, generates a complex sequence of multiple syscalls, most of them seemingly unrelated with the triggering of the high-level functionality. In this specific case, for instance, the complexity is due to the fact that the analyzed app ultimately has to invoke a series of syscalls belonging to different technical areas to complete this task, including inter-process communication with the system service that provides the trusted CA certificates, random-number generation for creating the nonce used to setup the connection, and network-related syscalls to perform the handshake with the remote server. While there are few works that reconstruct parts of this behavior (e.g., CopperDroid [30] focuses on reconstructing the semantics of specific activities, including Binder-related operations), it is not clear whether reconstructing this semantics gap is in fact possible or practical in the *general* case. In fact, even though it may be practical to scan for specific patterns in a sequence of syscalls, traces often contain thousands of syscalls that do not seem to relate to any common pattern.

Goal of this work. To date, we are not aware of any work that actually investigates the feasibility of reconstructing the high-level semantics from generic low-level syscall traces. The goal of this work is to fill this gap: *this paper presents the first systematic exploration of the challenges and feasibility of bridging the gap from trustworthy system calls to semantics-rich, but difficult-to-obtained high-level APIs.*

To this end, we have built a new analysis framework aiming at exploring the complexity of this research problem with a data-driven approach. The first key challenge is the scale: by dynamically analyzing 750 Android apps, we have collected data on over 40 million API invocations, which in turn generated over 13 milion syscalls invocation (interestingly, many API invocations do not invoke any syscall). We then process this low-level data to build a knowledge base of so-called "models," which aim at summarizing the big amount of raw data that we have collected in the previous step, to make it more viable for subsequent analysis. The complexity of the Android framework, the high number of exposed high-level APIs, the API's non-deterministic behavior, and the overlap generated by these APIs, make the analysis of this dataset far from trivial. To the best of our knowledge, this paper performs the first data-driven exploration of this problem space, and it provides evidence that this is a very difficult problem, significantly more challenging than what previously thought.

In summary, this work brings the following contributions:

- We systematically explored the research problem of *semantically lifting* a generic trace of performed system calls to a trace of invoked high-level APIs, with a focus on Android.
- We built a large-scale, annotated dataset that maps high-level APIs to the various "representations" of low-level syscall traces, and we provide an in-depth discussion of patterns and other interesting aspects.
- We develop and test different approaches attempting to perform the aforementioned semantic lift problem, and we show that this is a much more difficult problem than what previously thought.
- We provide recommendations and lessons learned that future work needs to consider when tackling this problem.

In the spirit of open research, we make our instrumentation framework, the collected dataset, and the analysis results publicly available at: `https://github.com/eurecom-s3/syscall2api` .

2 Background on Dynamic Analysis

Android framework API. Programming languages are commonly divided in two categories, depending on whether they provide a high- or a low-level abstraction over the computing system. High-level programming languages provide to the programmer a closer experience to a natural language and they are designed to perform complex tasks in few lines of code. On the other hand, low-level programming languages allow the programmer to interact with those aspects of computation that high-level programming takes for granted.

High-level programming languages expose to programmers a set of functionalities, called Application Programming Interface (API). In Android, these APIs are implemented in the so-called Android Framework. Some of these APIs are not implemented solely in Java, since their behavior exceeds the expressiveness of this language. They rely instead on the Java Native Interface (JNI), which provides a bridge toward parts of the framework written in C or C++. This is the case for some of the most complex and, arguably, the most security relevant APIs, like those that handle the personal data of the user, interact with the broadband, access the Internet, etc. Indeed, those functionalities require the intervention of the operating system to be accomplished. Being based on the Linux kernel, in the Android operating system a user space application can take advantage of the services exposed by the kernel by means of system calls (syscall from now on).

Even though it is true that any security sensitive operation is performed by means of syscalls, the contrary is not true. In fact, a vast number of syscalls are actually invoked to implement behaviors that are not strictly security-sensitive, such as user interaction, memory management, and thread synchronization.

It is important to note that not only the framework, but also apps can contain pieces of code written in low-level languages. Moreover, both the high- and low-level code run in the same process and with the same privileges and there is no security boundary between the two. This is a common misconception, which led previous works to overlook the role of native code in the realm of Android dynamic analysis [8].

Dynamic analysis. Understanding the behavior of a program is an important step toward determining whether it is malicious or not. Dynamic analysis aims to gather this information from running the program in a controlled environment, recording as much evidence of malicious activities as possible.

Depending on the type of the controlled environment, the collected information can vary. Execution traces are one of the most common types of evidence collected during dynamic analysis and they describe a timeline of what was executed in the context of the program under analysis. Different granularities are possible, including API- and syscall-level traces.

API tracing records all the high-level functions invoked during the execution. Different mechanisms have been proposed to obtain such traces, including framework modification, run-time hooking and Ahead-of-Time (AOT) compilation instrumentation. Unfortunately, all of them can be detected and evaded by native code components.

Framework modifications, for example, can be identified by a malicious application through memory introspection. Moreover these techniques rely on the assumption that the program uses the default run-time provided by the system, but a malicious application could ship its own run-time library as a native library, avoiding completely the instrumentation. Run-time hooking and AOT compilation instrumentation suffer from similar problems. They both assume that the malicious code is implemented by the app in the high level language. However, the malicious behavior could be perpetrated by the native code, for example by mimicking the same syscalls that the framework would invoke to complete the same task. More fundamentally, the fallacy of API tracing mechanisms resides in that the instrumentation is in the same security context of the program under analysis.

On the contrary, syscall traces can be obtained directly from the kernel, in a transparent way from the program perspective. There are several techniques to acquire syscall traces, the two most prominent being `strace` , a ptrace-based mechanism, and SystemTap [12], which inserts probes in kernel space and logs relevant information. The main drawback of syscall tracing is that the information collected are difficult to interpret. Finding evidence of malicious activity from a syscall trace alone can be a hard task.

3 Challenges

Reconstructing the semantic gap from a syscall trace is a task made particularly difficult by several challenges, which this section systematizes. We note that the discussion of these challenges is “conceptual” — a priori, it was not known whether these challenges would or would not actually pose problems when dealt with in practice. To the best of our knowledge, in fact, no previous work has ever explored the actual practicality issues that these challenges create. One of the contributions of this work is to fill this gap: as we will present throughout the paper, our experiments provide the first data-backed evidence that these challenges do cause profound problems.

Multiple possible execution paths. The first challenge is that different invocations of the same API could follow different execution paths. This could be the case for a number of reasons. An API could behave differently depending on the arguments with which it has been invoked. However, its behavior could also differ depending on the execution environment and context. Different execution paths of course imply that the number and type of syscalls that are executed upon API invocation can widely vary. For example, consider an HTTP-related API: from the perspective of syscalls invocations, the recorded trace can widely change depending whether the API’s argument is a valid URL, or whether the device has network connectivity. These aspects can clearly influence whether we would see network-related activity in the syscall traces.

Non-determinism. Another potential problem is non-determinism. With this term, we refer to those cases for which even if an API is invoked with the same arguments and within a “similar” environmental context, the syscall traces could still differ due to inherent non-determinism of the system or because of very subtle "internal" differences (e.g., the current internal state of the memory allocator). Naturally, one could argue that the lowest-level aspect of the system could be considered as part of the "context" and that this challenge is overlapping with the previous one. This would be, of course, a valid argument. Nonetheless, we opted to make this distinction explicit due to the different nature of the source of potential divergent behaviors. As we will discuss throughout the paper, the different nature greatly influences the *frequency* with which such non-determinism arise and *how* these problems should be tackled in practice.

Multiple layers of APIs. The Android framework is organized as a complex, multi-layer system of APIs: each API, especially the ones "exposed" to third-party apps, are implemented by invoking several others lower-level APIs. Indeed, it is rare that a high-level API directly invokes syscalls. This means that, when dealing with these higher-level APIs, every potential behavioral difference and non-determinism that affect lower-level APIs will be somehow combined—in a potentially combinatorial way. This makes capturing all different behaviors of a non-trivial API very challenging in the general case.

Inherent ambiguity of syscall traces. Different APIs often use the same syscalls to implement their behaviors. In other words, a given sequence of syscalls can (and often does) overlap across the execution of different APIs. From the perspective of analyzing a syscall trace to then understand which APIs have been actually invoked, this poses a significant challenge: it is very complex to "go back" with certainty as there are many different possibilities that may explain a particular sequence of syscalls. We then say that these syscall traces are *ambiguous* as it is often not possible to determine which, across a number of potential candidates, is the real API that has been actually invoked.

No clear boundaries. Given a syscall trace, it is challenging to determine when the syscall sub-trace of a specific API is starting or ending. In fact, there are no clear-cut markers signaling these aspects, and traces of different APIs (or even of the same one) may have different lengths. This aspect, together with the other aspects and the combinatorial nature of how low-level APIs are used to implement higher-level APIs, makes associating a series of syscalls to a given API much more challenging.

Event-based nature of Android apps. Android apps are written following an event-driven paradigm, which implies that apps often make use of callbacks. The classic example is the definition of an `onClick` callback to define what should happen when the user clicks on a specific button.

This pattern often causes several, nested control flow transitions from the Android framework to the Android app, and vice versa. In fact, consider what happens in the scenario where a user clicks on a button: 1) the control flow transitions from the framework to the `onClick` callback method, implemented in the app; 2) the `onClick` method may invoke several Android APIs, which would cause the control flow to transition back to the Android framework; 3) when the execution of these APIs is over, the control flow goes back to the `onClick` method; 4) when the `onClick` method ends its execution, the control flow goes back to the Android framework once again. These various control flow transitions make our analysis significantly more complicated. We note that these problems do not affect more traditional programs that do not heavily rely on asynchronous callbacks.

APIs cannot be invoked without proper context. With the aim of collecting data about which syscalls are invoked by which API, one possibility would be to consider each API separately and automatically invoke it within an instrumented environment. Unfortunately, this approach would not work in practice. In fact, the vast majority of APIs need to be invoked with the appropriate context, or otherwise they would quickly quit their execution due to errors. Moreover, many of these APIs require a proper "receiving" object to be invoked on, and the automatic creation of such objects is a very challenging task per-se.

4 Approach

This section discusses how we approached the various challenges discussed in the previous section. Our approach is summarized in Figure 1.

The first step of our approach consists in building a dataset containing *which syscalls are invoked by which API*. Conceptually, the idea is to use this dataset as a sort of ground truth, to then use it to perform additional experiments. As mentioned earlier, this task is challenging per-se. In fact, we cannot just create code to execute the various APIs, as we would not know with which arguments we would need to invoke them and from which context. We approached this problem by taking a large number of *benign* Android apps and by executing them in an instrumented environment to produce both API- and syscall-level traces. This step is discussed in Section 5.

This raw data is sparse and contains a remarkable amount of redundancy, and the scale of this data does not make it suitable to be used for additional analysis without a preprocessing step. Thus, in a second step, the raw data is then organized in a data structure (that we refer to as *knowledge base*), which lays the foundation for subsequent analysis. For this step, the idea is to eliminate unneeded redundancy and to somehow obtain a concise representation of what contained in the dataset. The output of this analysis is a set of so-called *API models*. These models offer a "usable" over-approximation of all the behavior collected during the initial analysis phase (see Section 6).

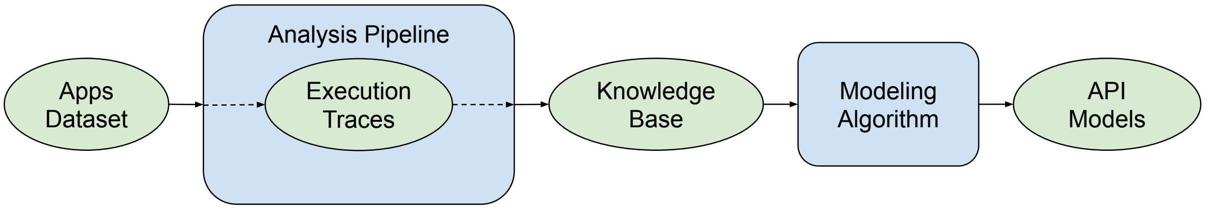

Figure 1: Overview of the approach.

The specifics of these API models have been designed to be useful for two different purposes. First, we perform the first empirical data exploration on this peculiar dataset (see Section 7), and we use it to uncover patterns and high-level metrics that show how challenging the problem of semantics reconstruction actually is. Second, we use these API models to take the first steps toward *mapping a generic sequence of syscalls to their associated APIs*, as discussed in Section 8.

5 Knowledge Base

```
API A: Entering
  Syscall w
  Syscall x
  API B: Entering
    Syscall y
  API B: Exiting
  API C: Entering
    Syscall z
  API C: Exiting
API A: Exiting
```

Listing 1: Example of an analysis trace.

In this section we present the methodology we followed to create our knowledge base. We started by considering a set of *benign* Android apps, which we then analyzed within our analysis framework, discussed in this section. This analysis framework consists in an instrumented environment capable of logging traces of *both* syscalls and APIs. These raw analysis traces are then parsed and loaded in a more suitable tree-based data structure.

5.1 Analysis Tracing Pipeline

Syscall-level tracing. To log the syscalls invoked by a given app we relied on `strace` , which is a robust, off-the-shelf tool based on the `ptrace` syscall. For each syscall, we traced the timestamp, the calling *thread id*, the syscall name and its arguments. For obvious performance reasons, this behavior can be selectively enabled on the application under analysis only, so to avoid to slow the entire system down with unneeded instrumentation.

We note that, in principle, relying on strace has two disadvantages. First, it can be detected by an app. While this is true, this is not a problem at this stage because our goal is to collect the behavior of benign apps, which we assume to not contain anti-debugging techniques. Moreover, strace can be completely implemented in kernel space [12,20], making it more resilient to anti-debugging techniques and suitable for the analysis of malicious programs. Second, strace can cause a significant slowdown. However, once again, this is not a significant concern in our scenario as we are analyzing apps to collect "as much behavior as possible," and we do not necessary need to cover "all" the behavior of an app. In other words, while the slowdown may make us lose some behavior, this aspect does not threaten the validity of our experiments. The aspect that is actually of critical importance is that all the events (both syscalls and APIs) are logged in the appropriate chronological order, which is the case for our system.

API-level tracing. To log the APIs invoked by a given app, we first considered well known instrumentation frameworks, such as Xposed [6] and Frida [2]. In fact, one of the main features of these frameworks is the possibility of tracing specific API methods. Unfortunately, it turns out that when attempting to hook more than a few hundreds APIs, these frameworks make the system unstable, leading to repeated crashes. This is a problem as the Android framework is constituted by tens of thousands of APIs.

To this end, we have developed a new solution, which is based on source code instrumentation. By means of JavaParser [4], we automatically instrumented *all* the public methods in the AOSP framework. In particular, we added a call to `logApi()` — a new static method that we defined in the `java.logging.Logger` class — at the entry point and at all exit points (e.g., return statements, catch blocks of exceptions) of every instrumented method.

The `logApi` method takes a string as its first argument, which our instrumentation pass uses to specify *which* API has been invoked. In particular, this string contains the name of the instrumented method and whether the call originates from an entry point or an exit point (i.e., whether the method has been just invoked or whether its execution is about to end). Under the hood, this `logApi` method simply invokes a `write` syscall, using as an argument the same argument received by the `logApi` method itself.

This technical solution gives us a setting where both syscalls and APIs logging converge in the same *unified tracing channel*. Since these analysis traces are also thread-aware (by simply logging the thread id of the thread that invoked the API or syscall), all the log entries are already chronologically ordered and consistent, by design.

Example of an analysis trace. The result of this step is a merged analysis trace, which contains both syscalls and APIs, with the corresponding thread id and timestamp. Listing 1 shows an example of an analysis trace. In the listing, it is possible to see how the system can transparently log both *enter* and *exit* events for both syscalls and APIs.

5.2 Building a Knowledge Base

The analysis traces created in the previous step contain *all* the information collected, but they are not easily processable. To this end, we post-process these traces and we organize them in a more suitable data structure. This structure consists in a key-value store in which each key is the fully qualified method name of an API, and each value is a list of entries, each of which represents a specific instance of an API invocation. Each of these entries contains a list of events recorded between the start and the end of that specific instance of the API invocation. The events can either be "syscall invocation" or "API invocation."

6 API Models

Invocations of the same API usually share common features but they are not always completely identical. For example, the two different branches of an if-else construct in the API code can lead to different sequences of API calls or syscalls (see Section 3 for a more systematic discussion of similar challenges). The knowledge base discussed in the previous section contains all relevant information and it can be already used as a source of interesting data. However, it cannot be used to recover the high-level semantics without some kind of pre-processing. The reason for this is that APIs can potentially have a big number of different invocations (see Section 7) and each invocation can be significantly different from others.

In this section we introduce the concept of *API models*, their design and the rationale behind it. We then present an algorithm that creates API models starting from the invocations of an API. The last part of this section discusses how a sequence of syscalls can be then matched against these API models.

6.1 Anatomy of an API Model

In the context of this paper, an API model is an object that summarizes the common features between different invocations of the same API. A model is constituted by an ordered sequence of symbols representing the various APIs and syscalls found in the invocations. Each symbol in the model can have an optional *modifier* that indicates that it can appear up to an unlimited number times.

API models represent an over-approximation of all the information stored in the knowledge base. In fact, by assuming that a syscall pattern can be repeated up to an unlimited number of times, an API model can match sequences that have not been observed in the API invocations.

The choice to model repetitions of syscalls in this way is driven by the intuition that repeated patterns generate from loops in the execution. Those loops can be repeated either a fixed or a variable number of times. Our API models make use of the repetition modifier only if the same pattern has been observed repeating itself a different number of times in distinct invocations.

6.2 API Models Creation Algorithm

The model creation phase consists in applying a two-step algorithm to all the APIs in the knowledge base. In the first step we identify all those invocations that are identical according to the following definition: *two invocations are identical if they contain the same API calls and syscalls in the same order*. Note that the syscalls arguments are not taken into account. Duplicate invocations are not taken into account from further processing.

In the second step the algorithm processes each of the remaining invocations to create a list of models for each API. In particular, the algorithm proceeds as follows. It first attempts to find the longest repeated pattern in the invocation under analysis. If a repeating pattern is found (e.g., a single syscall or a sequence of syscalls that keep repeating itself), the algorithm creates a model similar to the original invocation, except for the repeating pattern that is marked with the repetition modifier. This model is then checked against the other invocations. If at least one of them produces a match, the generalization is considered "useful" and the model is added to the list of API models. If not, it means that this over-generalized model was not useful: the algorithm thus discards it, and adds to the list of API models the trivial model matching the "exact" invocation under analysis.

6.3 API Models Matching

Once these API models are computed, the next step is to approach the *mapping problem*. Given a sequence of syscalls, this problem aims at determining which API is the most likely to have generated such a sequence. In a way, the goal is to *map* a given sequence to the "correct" API. There are of course many different possible algorithms and strategies to implement this.

For this paper, we opted to implement two *extreme* strategies: the longest match and shortest match strategies. In both cases, the algorithm starts from the very beginning of the syscall sequence. It then considers all the API models in our database and it determines which of these API models actually match the given sequence of syscalls. The algorithm then selects the longest (or the shortest) of these matches, and this initial sequence of syscalls is considered as covered. The algorithm then proceeds by applying the same method to the sequence of syscalls that followed the one that is covered by the selected API model.

We note that this constitutes the first step into reconstructing the API trace starting from a sequence of syscalls. We present an evaluation of these two strategies in Section 7. Of course, we acknowledge that there are in fact many other potential strategies. However, we believe that considering these two extreme strategies is a promising first step toward understanding and exploring this relevant research problem.

7 Data Exploration

This section explores our knowledge base (KB) by discussing interesting statistics and insights. We start by giving more precise information about the apps that we analyzed for collecting the analysis traces and the experimental setup. We then present measurements about the information stored in the KB. These measures provide empirical data highlighting the difficulties in reconstructing the high-level semantics from generic low-level syscall traces. Specifically, we will show how the APIs in the KB are very diverse and, because of their nature, how different APIs present different challenges for semantic reconstruction. We will also show how a human analyst can query the KB and how this is important in highlighting the problematic nature of two aspects, namely noise and ambiguity, making semantic reconstruction a task more challenging than what previously thought. We then explore these two aspects in an automated fashion and we discuss the gained insights in Section 7.3 and 7.4, respectively.

7.1 Apps Dataset and Experimental Setup

As mentioned throughout this paper, we opted for a data-driven approach to explore the problem of semantics gap reconstruction. We build our ground truth of analysis traces by recording the execution of a set of 750 apps. We collected these apps from the F-Droid Open Source Android App Repository [1]. In particular, we selected *all* apps from this dataset that used at least one dangerous permission. The rationale behind this choice is that these apps would tend to be more complex than others not requiring any permission, and would thus have more chances to expose interesting behavior.

	Leaf APIs	Non-Leaf APIs
Empty APIs	1730	-
Monoform APIs	29	810
Multiform APIs	573	1488

Table 1: API occurrences in KB. Note: there cannot be Empty APIs that are also Non-Leaf APIs

	Leaf APIs	Non-Leaf APIs
Empty APIs	2850	-
Monoform APIs	59	665
Multiform APIs	94	962

Table 2: API occurrences in KB (after noise reduction)

Each app was executed for five minutes on a Google Nexus 5X device, which was previously instrumented with our modified Android framework, as described in Section 5.1. We then used the Android Monkey Runner [16] to stimulate the app's user interface. We fully acknowledge that the Monkey Runner is not sophisticated and may not trigger deep parts of the app's codebase. However, we note that the rationale behind these experiments is not to fully cover a specific app, but to execute many apps and collect what we can from each of them.

7.2 API Classification and Statistics

Our KB contains invocations for a total number of 4,630 distinct APIs. The total number of API invocations observed is over 40 million, while the total number of syscall that these API invocations invoked is over 13 milion. Note that the number of API invocations is larger than the number of syscalls, which means that a significant number of API invocations do not result in any syscall. In average, each API has been observed 8,721 times, while the average number of events in the invocation lists is 0.84 (3.63 without considering empty invocation lists).

We categorize the APIs according to two different aspects. First, we consider how many different entries each API has in its invocation list. That is, for each API we look at how many different syscall sequences we have recorded in our dataset. We distinguish three different cases: 1) *Empty APIs*, those whose invocations are all empty lists; 2) *Monoform APIs*, those for which all the invocations are equals (and non-empty); and 3) *Multiform APIs*, those that have at least two different non-empty invocations. Second, we cluster APIs according to the *type* of events that each of their invocations contains. For this aspect, we distinguish between 1) *Leaf APIs*, those whose invocations contain only syscalls, and 2) *Non-Leaf APIs*, those containing at least one API in their invocation lists.

Table 1 shows the occurrences of each category of APIs in our KB. It is interesting to understand how each category of API plays a different role in the context of semantic reconstruction. *Empty APIs* are those that are completely implemented in user space and do not make any system calls. For this reason, their behavior cannot be identified at all based on syscall information only. *Multiform APIs* make the overall task of semantic reconstruction very challenging because all of their invocations must be taken into account. Our modeling algorithm, for example, could produce more than one model for each API in this class. *Monoform APIs*, on the other hand, are simpler because their only invocation can also be used as model.

Leaf APIs are the closest to syscalls in terms of semantics. Some of them invoke always the same syscall (e.g., *android.net.LocalSocket.setSoTimeout* always executes the syscall *setsockopt*). They are also the easiest to reconstruct, since their behavior can be recognized directly from the executed syscall. To reconstruct a *Non-Leaf API*, instead, one needs first to reconstruct the other APIs that it could invoke.

7.3 Noise Patterns Identification

While inspecting the data offered by our knowledge base, we came across surprising insights. For example, we found some syscalls in the invocation list of a few of those APIs that were expected to be *empty*. One example is the *java.lang.StringBuilder.append* API. Interestingly, while the vast majority of the invocations of these APIs were indeed empty, (very) few of these invocations contained peculiar syscall patterns that, at first glance, we could not explain with the expected behavior of the API. To our surprise, we then found that these patterns were also observed in the models built for *other* APIs.

To investigate this unexpected finding, we developed a post-process analysis pass to automatically identify similar cases. The key idea is to perform anomaly detection. Our system identifies a model of an API as an outlier if the model describes a number of invocations (of that API) that is lower than a certain threshold (for our tests, we used $1/1000$ as a threshold). With this tool, we identified three "noise" patterns. The first relates to thread synchronization (e.g., *futex*, *sched_yield* and *clock_gettime* syscalls). The second relates to memory management (e.g., *madvise* and *mprotect*) or their combinations. Finally, the third pattern we identified relates to the specific malloc implementation in Android's *bionic* C library: since it is used to obtain memory for the allocation of new objects, it can potentially appear during the invocation of any API that allocates Java objects—and, similarly to the other two cases, this is what causes the noisy pattern.

To better explore the role that these noisy patterns have in our dataset, we opted to eliminate from our models all the syscall patterns that contribute to such noise (i.e., the ones mentioned above), since we believe that these classes of syscalls do not carry any meaningful information that can be used to reconstruct any API semantic. Table 2 shows statistics for each class of APIs in the KB after removing the noisy patterns. A comparison of these data with those in Table 1 suggests that noise reduction leads to more consistent data. For instance, the number of APIs that have two or more different invocations decreased by almost 50%.

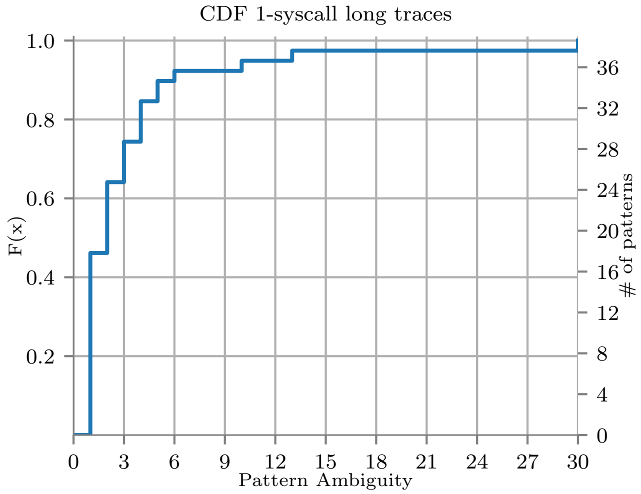

Figure 2: Pattern Ambiguity for 1-syscall long patterns

7.4 Ambiguity Measurement

Another aspect we explored relates to the inherent ambiguity of models included in our KB. For example, we noticed that some models overlap or are identical, even though they belong to different APIs. This means that potentially more than one API model can provide a match for the same syscall pattern, leading to ambiguity in the results of any matching algorithm.

With the goal of quantifying the ambiguity of the results in our dataset, we define a new metric, which we call *ambiguity score*. This metric is an integer number that can be computed for *each pattern of syscalls*. We define this metric as the number of different APIs (in our KB) that match against a given pattern of syscalls. We tackle this problem by considering syscalls patterns of different lengths. Moreover, we consider two different values: *pattern ambiguity score* and *total pattern ambiguity score*, the only difference between the two being that in the latter case we weight a pattern according to how many times it appeared in our traces. Figure 2 and Figure 3 show the cumulative distribution functions (CDF) of the ambiguity score for 1-syscall and 2-syscall long patterns respectively (after having removed the noise). Figure 4 and Figure 5, instead, plot the CDF of the *total* ambiguity score, for the same patterns. These figures show the data before and after removing the noise. For a better visual comparison be-

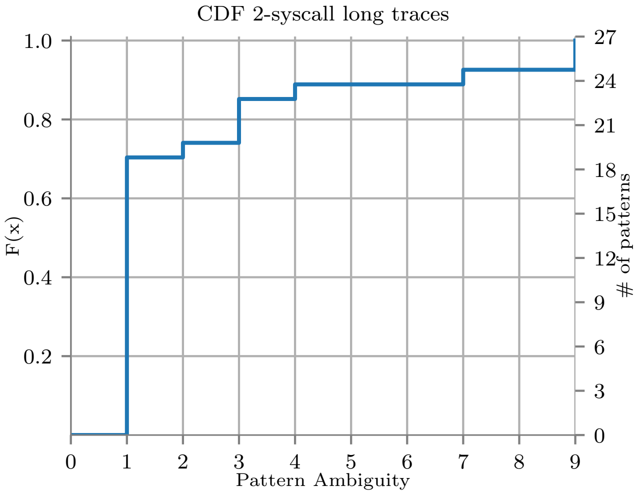

Figure 3: Pattern Ambiguity for 2-syscall long patterns

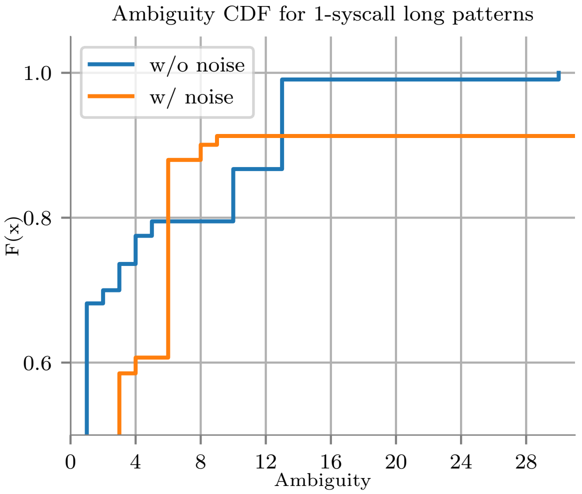

Figure 4: Total Pattern Ambiguity Comparison (1-syscall long patterns)

	Shortest Match		Longest Match	
	w/ noise	w/o noise	w/ noise	w/o noise
Trace coverage	26.4%	45.2%	33.0%	61.5%
Correct matches	30.9%	38.9%	26.4%	46.9%

Table 3: Results of two variants of the matching algorithm.

	Method	Class	Package
Trace Coverage	61.8%	62.0%	63.1%
Correct Matches	46.9%	47.0%	49.3%

Table 4: Accuracy results under relaxed definition of correctness.

tween the CDFs, the figures show the data around the point in which the CDF of the noiseless KB reaches 1.0. The CDF of the noisy KB instead reaches 1.0 at much higher abscissa (not shown in the figures), since it raises at a slower pace with respect to those of the noiseless KB. This is also true for the CDFs built for other syscall pattern lengths, meaning that models built without removing the noise are more ambiguous than their noiseless counterpart. For the interested reader, we also report, in the Appendix, the CDFs for patterns of different lengths (from three to up to five).

8 Exploring the Mapping Problem

This section discusses a first attempt to reconstruct the semantics gap of a *generic* sequence of syscalls. The input to this step is a non-annotated syscall trace and we investigate how two strategies would perform in this context. The challenges discussed in Section 3 make this task particularly difficult.

To this end, we define the notion of correct match as follows. A match is *correct* if it spans the same syscalls of an API in the annotated trace and if said API is in the set of those that the algorithm selected as candidates. This means that a match is not considered correct if, for example, it covers more syscalls than the ones actually produced by the API, or if it does not start exactly on its first syscall.

We measure the results of the reconstruction process in terms of percentage of APIs correctly identified (i.e., the ratio between the number of APIs correctly identified and the total number of APIs in the annotated trace) and percentage of the traces covered by correct matches (i.e., the ratio between the number of syscalls correctly assigned to a candidate API and the number of syscalls effectively in the trace).

We implemented two variants of a matching algorithm: a *shortest match* and a *longest match* policy. Table 3 reports the results in terms of *trace coverage* (i.e., which percentage of the trace was possible to cover) and *correct matches* (i.e., the ratio of matches that are correct). These results show that, clearly, the “longest match” heuristic results are better, as it produces higher rates of correct matches and trace coverage in each single test. Table 3 also provides a comparison of the results obtained by adopting the models built before and after noise reduction. It is interesting to note that not only noise reduction decreases the ambiguity (as shown in Section 7.3), but it also increases the amount of correct matches.

We note that the results reported thus far use a quite aggressive definition of “correctness.” In a way, we consider a match correct if and only if the algorithm is able to identify the specific, exact API. Since there are cases in which different APIs belonging to the same class (or package) actually have the same semantics, we decided to explore how the accuracy would change under a more relaxed definition of “correct match.” Table 4 shows the percentage of trace coverage

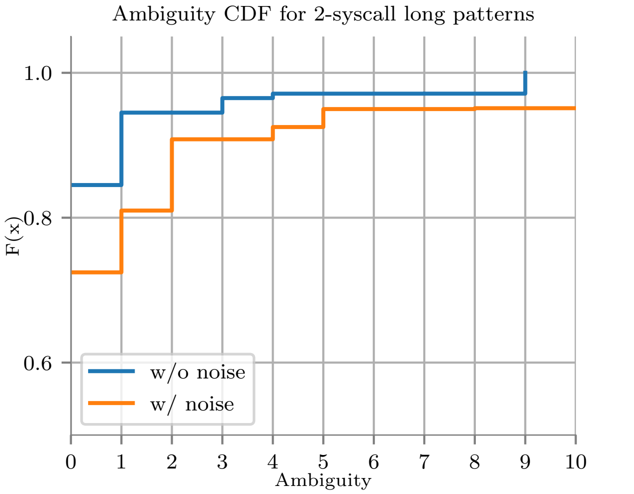

Figure 5: Total Pattern Ambiguity Comparison (2-syscall long patterns)

and match correctness when a match is considered as correct in three different scenarios (using the "longest match" heuristics and after noise removal): we consider a match correct if the method name, the class name, or the package name of the API matched (instead of its fully qualified name, which also contains the types of its arguments). The table shows that the numbers do improve, but that they are still far from ideal. We believe that the reason for the low accuracy of the results of our algorithm resides in the fact that it fails in recovering from an incorrect match caused by a length mismatch. In this situation, the algorithm is out of synchronization as it tries to match the next API from the very next syscall in the trace.

Another source of desynchronization is given by those sections of the trace in which no APIs are recorded (i.e., when the execution of the application returns to the framework, see Section 3 for more details). This issue cannot be solved applying the same approach used for APIs, but they do not share enough features to build meaningful models. Moreover, these areas contain similar patterns to those observed in API models, making it difficult to distinguish between them.

Discussion. These results show that simple strategies are far from enough to properly map sequences of low-level syscalls to high-level APIs. The mapping task appears even more challenging when considering that our experimental setup made the analysis, in theory, much simpler than what it would be in a real scenario. In fact, our experiments attempt to map sequences of syscalls to APIs that have been extracted from the same knowledge base. In a real scenario, instead, the algorithm would not have any guarantee that the potential target API is one API already in the knowledge base (as an unknown app may make use of APIs never seen in the training set). Moreover, we even simplified the problem by assuming that the starting point of the first API in the trace is known. Last, our "correctness" definition is quite generous, as it considers an API match as correct if the correct API is among one of the selected candidates: ideally, the perfect matching system should indicate only one API for each match. We believe these observations strength our key hypothesis: *that, even under these very favorable conditions, the mapping problem presents inherent difficulties that are very challenging to overcome*.

Future directions. We believe that reducing the noise in the models is a key component for a successful approach to the mapping problem, since we identified that the noisy patterns produced by the framework and the ART runtime a strong source of ambiguity. To this aim, we believe that future work should investigate a more aggressive noise reduction strategy to improve the results. Another direction for improvements could be to leverage the fact that some APIs are often used in conjunction with others, providing a heuristic for choosing between multiple candidate APIs.

At the moment, our system collected information about the APIs exposed via Java public method. A different approach would be to monitor the invocation of every single Java method (including private ones) in the framework. On the one hand, this approach could shed light on those areas of the traces that seemingly do not contain any API. On the other hand, private methods are more difficult to interpret for an analyst since they are undocumented and require knowledge of the internals of the Android framework to be fully understood.

Another step of our pipeline that can be enhanced is the model creation algorithm. In general, we believe that future works should focus on creating API models that describe as many features of the API invocations as possible. In this work we show how to model repeating patterns, but there are other features that are worth modeling. For example, one possible improvement can be to summarize in the same model all those invocations that differ for a small number of syscalls and/or API calls only.

In an attempt to model this type of scenarios, a first version of our prototype relied on the Needleman-Wunsch sequence alignment algorithm [26]. The rationale was that by aligning two invocations is possible to find those syscalls and API calls that appear in only one of them. We leveraged the aligned sequences to create models in which the symbols corresponding to these API calls and syscalls were marked with an additional modifier. This modifier indicates that the symbol to which it is applied is "optional," meaning that the model matches a sequence regardless of its presence. This approach, however, led to unacceptably high computational complexity of the pattern matching phase, to the point of making it unfeasible in practice for models with many entries. Still, we believe there could be some value in adopting this technique for only a subset of the APIs, especially those whose invocations contain only a small number of syscalls and API calls.

Lastly, we believe our approach would clearly benefit from integrating the results from existing systems like CopperDroid [30], which already perform several steps to recover the semantics of the *ioctl* syscalls. This particular class of syscalls is among the most frequent and ambiguous ones in our knowledge base. This is due to its fundamental role in the low-level implementation of the Binder subsystem, which relies on the *ioctl* syscall to exchange "parcels" of data between user apps and system services. Integrating CopperDroid with our system would add the corresponding Binder semantics to each *ioctl* syscall, which would eventually reduce the ambiguity for this class of syscalls significantly.

9 Related Work

The Android security community has published a vast number of works related to program analysis of unknown apps. This section places our work in the context of two main related areas, namely static and dynamic program analysis.

Several static analysis approaches have been proposed to analyze Android apps, and malware in particular. Some of the early works in this area include RiskRanker [18] and DroidRanger [34], which rely on symbolic execution and a set of heuristics to detect unknown malicious applications. Another work is Apposcopy, which uses a signature-based approach to detect known malware samples [14]. Other works do not only focus on malware detection, but are more generic and attempt to identify suspicious data flows via taint analysis. Two relevant works in this area are FlowDroid [10] and DroidSafe [17].

Another important trend of works attempts to perform malware classification by using machine learning techniques. Some of the early works include Drebin [9] and DroidAPIMiner [7], which both extract several features from Android applications (e.g., requested permissions, invoked framework APIs) and then apply machine learning techniques to perform classification. A different system is AppContext [32], which uses machine learning techniques to identify malware by using the "context" of each behavior as a feature. More recently, Mariconti et al. proposed MaMaDroid [23], a tool that uses Hidden Markov Model chains and, once again, starts from the API function calls to build behavioral models. Another recent work in a similar direction is SLAP [22], which also uses machine learning with features based on API-related information, with the difference that it attempts to be more resilient to adversarial samples.

There has also been extensive research on program analysis of Android apps through dynamic analysis. Enck et al. [13] present TaintDroid, a dynamic taint analysis that performs whole-system data flow tracking through modifications to the underlying Android framework and native libraries. Other efforts, such as Mobile Sandbox [29] and Andrubis [21], developed tools and techniques to dynamically analyze unknown Android applications. Another trend of works has proposed approaches based on dynamic analysis to perform multipath execution and dynamic symbolic execution on Java and Android applications [5, 15, 25, 27, 31]. These approaches achieve higher code coverage than simpler dynamic analysis tools.

We note that *all these existing works use API-level information as the main building block for their analysis*. Their main rationale is that API-level information provides semantics-rich data, which in many cases is enough to discern benign apps from the malicious ones. This common trait of all these recent works underline the importance that API-related, semantics-rich data can play within the Android security research community. However, as often mentioned in this paper, all these approaches can be detected and evaded [8].

One of the very few works that fully acknowledge this limitation and performs a step forward is CopperDroid [28, 30]. In this work, the authors show how it is possible to reconstruct two categories of high-level behaviors. The first one consists in those implemented through Android Services, which CopperDroid identifies by unmarshaling objects used in Binder transactions (e.g., access to geolocalisation). The second includes those behaviors that result in a sequence of syscalls with a clear data dependency, which CopperDroid reconstructs by means of a value-based data flow analysis technique (e.g., opening a file and performing operations on it).

However, the authors of CopperDroid also note that API-level information, while useful in reconstructing high-level behaviors in some cases, is not fully trustworthy when dealing with some complex scenarios. Take, as an example, the behavior of the *createSocket* method of the *SSLSocketFactory* Java class. This API creates an SSL tunnel over an already opened TCP socket. Exploring our dataset we noticed that to perform this task the framework invokes various syscalls, for example, "getrandom" to generate the nonce used for the encryption or "read/write" to perform the handshake. By simply inferring the data dependency between syscalls, CopperDroid would be able to recognize its network-related aspect, but it would fail in understanding that the syscall pattern is actually implementing a tunnelling mechanism that, according to the API documentation [3], enables to instantiate an SSL connection over a proxy.

Our work thus differs from CopperDroid by exploring the behavior reconstruction problem in a more generic way: given a list of syscalls, is it possible to build a pattern identification and "go back" from syscall to API in the general case? In other words, in this paper we are interested in answering a more generic question, and we do not rely on specific patterns or on data-dependency among syscalls to address the mapping problem. The main difference with previous works is that we focused on using a data-driven approach to explore the more-generic problem of performing semantics reconstruction of a generic sequence of syscalls. In the process, we have also built the first dataset of API-syscall relationship — to the best of our knowledge, the first of its kind.

The problem of reconstructing high-level behaviors from low-level features is not exclusive to the Android security research field. Martignoni et al. [24], for example, modeled a set of malicious behaviors found in different malware families for the Windows operating system. To this aim the authors manually analyzed the executions traces of malware and benign programs to express high-level behaviors in terms of lower-level syscall-like events.

The main drawback of their approach is that the modeling phase cannot be automated because it requires human understanding of each high-level behavior. In our work, instead, the modeling phase is completely automated. Moreover, our approach is more generic since it models more than just a restricted set of behaviors and is not bound to malicious behaviors only.

10 Conclusion

Dynamic analysis has proven to be fundamental in the field of program analysis, especially when it provides high-level and human-friendly information about a program's behavior, like in the case of API traces. As discussed, a vast number of research works rely on these semantics-rich API traces. Unfortunately, all current instrumentation frameworks can be detected and evaded, making these techniques not suitable to be used in an adversarial context, like malware analysis.

This paper provides the first systematic exploration on the challenges of automatically reconstructing API traces semantics from low-level syscall traces. Ideally, a system that can complete this task in an accurate manner would provide all the advantages of API semantic analysis, while remaining undetectable by malicious programs.

The evaluation of this work shows that this task is challenging — arguably much more challenging than what previously thought. While previous works focus on recognizing specific patterns, we adopt a generic data-driven approach and we collected and analyzed data on several millions API and syscall invocations. As one of the core contributions, we built a new dataset and an analysis pipeline, which we publicly release to encourage future work in this important area.

Acknowledgements

We would like to thank the anonymous reviewers for their valuable feedback. We are also grateful to our shepherd Lorenzo Cavallaro for his suggestions, and Andrea Possemato for helping with some experiments. There are several others we are also in debt with: Carlo, Harvey, Ivan, Luca, Lorenzo, and, it goes without saying, Betty Sebright. Thanks to all of you.

This material is based upon work supported by the NSF under Award number CNS-1849803. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the NSF.

References

[1] F-droid — free and open source software applications for the android platform. https://f-droid.org/ .

[2] Frida analysis framework. https://www.frida.re .

[3] Java documentation for javax.net.ssl.sslsocketfactory.createsocket. https://docs.oracle.com/javase/8/docs/api/javax/net/ssl/SSLSocketFactory.html#createSocket-java.net.Socket-java.lang.String-int-boolean- . Accessed: 2019-06-27.

[4] Java Parser and Abstract Syntax Tree for Java. https://github.com/javaparser/javaparser .

[5] JPF-symbc: Symbolic PathFinder. http://babelfish.arc.nasa.gov/trac/jpf/wiki/projects/jpf-symbc .

[6] Xposed installer (framework). http://repo.xposed.info/module/de.robv.android.xposed.installer , 2017.

[7] Yousra Aafer, Wenliang Du, and Heng Yin. Droidapiminer: Mining api-level features for robust malware detection in android. In *Proceedings of the Security and Privacy in Communication Networks (SecureComm)*, 2013.

[8] Vitor Afonso, Antonio Bianchi, Yanick Fratantonio, Adam Doupé, Mario Polino, Paulo de Geus, Christopher Kruegel, and Giovanni Vigna. Going native: Using a large-scale analysis of android apps to create a practical native-code sandboxing policy. In *The Network and Distributed System Security Symposium*, pages 1–15, 2016.

[9] Daniel Arp, Michael Spreitzenbarth, Hubner Malte, Hugo Gascon, and Konrad Rieck. Drebin: Effective and Explainable Detection of Android Malware in Your Pocket. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2014.

[10] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*, 2014.

[11] Marko Dimjašević, Simone Atzeni, Ivo Ugrina, and Zvonimir Rakamaric. Evaluation of android malware detection based on system calls. In *Proceedings of the 2016 ACM on International Workshop on Security And Privacy Analytics*, pages 1–8. ACM, 2016.

[12] Frank Ch Eigler and Red Hat. Problem solving with systemtap. In *Proceedings of the Ottawa Linux Symposium*, volume 2006, 2006.

[13] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *Proceedings of the USENIX Conference on Operating Systems Design and Implementation*, 2010.

[14] Yu Feng, Saswat Anand, Isil Dillig, and Alex Aiken. Apposcopy: Semantics-Based Detection of Android Malware Through Static Analysis. In *Proceedings of the ACM Symposium on the Foundations of Software Engineering (FSE)*, 2014.

[15] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: directed automated random testing. In *ACM Sigplan Notices*. ACM, 2005.

[16] Google. UI/Application Exerciser Monkey | Android Developers. `http://developer.android.com/tools/help/monkey.html` .

[17] Michael Gordon, Deokhwan Kim, Jeff Perkins, Limei Gilham, Nguyen Nguyen, and Martin Rinard. Information-Flow Analysis of Android Applications in DroidSafe. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2015.

[18] Michael Grace, Yajin Zhou, Qiang Zhang, Shihong Zou, and Xuxian Jiang. RiskRanker: Scalable and Accurate Zero-day Android Malware Detection. In *Proceedings of the International Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2012.

[19] Roee Hay, Omer Tripp, and Marco Pistoia. Dynamic detection of inter-application communication vulnerabilities in android. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, pages 118–128. ACM, 2015.

[20] Tobias Holl, Philipp Klocke, Fabian Franzen, and Julian Kirsch. Kernel-assisted debugging of linux applications. In *2nd Reversing and Offensive-oriented Trends Symposium 2018 (ROOTS)*, November 2018.

[21] Martina Lindorfer, Matthias Neugschwandtner, Lukas Weichselbaum, Yanick Fratantonio, Victor Van Der Veen, and Christian Platzer. Andrubis–1,000,000 apps later: A view on current android malware behaviors. In *2014 Third International Workshop on Building Analysis Datasets and Gathering Experience Returns for Security (BADGERS)*, pages 3–17. IEEE, 2014.

[22] Aravind Machiry, Nilo Redini, Eric Gustafson, Yanick Fratantonio, Yung Ryn Choe, Christopher Kruegel, and Giovanni Vigna. Using Loops For Malware Classification Resilient to Feature-unaware Perturbations. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, 2018.

[23] Enrico Mariconti, Lucky Onwuzurike, Panagiotis Andriotis, Emiliano De Cristofaro, Gordon Ross, and Gianluca Stringhini. Mamadroid: Detecting android malware by building markov chains of behavioral models. *arXiv preprint arXiv:1612.04433*, 2016.

[24] Lorenzo Martignoni, Elizabeth Stinson, Matt Fredrikson, Somesh Jha, and John C Mitchell. A layered architecture for detecting malicious behaviors. In *International Workshop on Recent Advances in Intrusion Detection*, pages 78–97. Springer, 2008.

[25] Nariman Mirzaei, Sam Malek, Corina S. Pasreanu, Naeem Esfahani, and Riyadh Mahmood. Testing Android Apps Through Symbolic Execution. In *ACM SIGSOFT Software Engineering Notes*, 2012.

[26] Saul B Needleman and Christian D Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of molecular biology*, 48(3):443–453, 1970.

[27] Siegfried Rasthofer, Steven Arzt, Marc Miltenberger, and Eric Bodden. Harvesting Runtime Values in Android Applications that Feature Anti-Analysis Techniques. In *Proceedings of the Annual Symposium on Network and Distributed System Security (NDSS)*, 2016.

[28] Alessandro Reina, Aristide Fattori, and Lorenzo Cavallaro. A system call-centric analysis and stimulation technique to automatically reconstruct android malware behaviors. *EuroSec, April*, 2013.

[29] Michael Spreitzenbarth, Felix Freiling, Florian Echtler, Thomas Schreck, and Johannes Hoffmann. Mobile-sandbox: having a deeper look into android applications. In *Proceedings of the ACM Symposium on Applied Computing (SAC)*, 2013.

[30] Kimberly Tam, Salahuddin Khan, Aristide Fattori, and Lorenzo Cavallaro. CopperDroid: Automatic Reconstruction of Android Malware Behaviors. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2015.

[31] Michelle Y Wong and David Lie. Intellidroid: A targeted input generator for the dynamic analysis of android malware. In *NDSS*, volume 16, pages 21–24, 2016.

[32] Wei Yang, Xusheng Xiao, Benjamin Andow, Sihan Li, Tao Xie, and William Enck. AppContext: Differentiating Malicious and Benign Mobile App Behaviors Using Context. In *Proceedings of the International International Conference on Software Engineering (ICSE)*, 2015.

[33] Akira Yokoyama, Kou Ishii, Rui Tanabe, Yinmin Papa, Katsunari Yoshioka, Tsutomu Matsumoto, Takahiro Kasama, Daisuke Inoue, Michael Brengel, Michael Backes, et al. Sandprint: fingerprinting malware sandboxes to provide intelligence for sandbox evasion. In *International Symposium on Research in Attacks, Intrusions, and Defenses*, pages 165–187. Springer, 2016.

[34] Yajin Zhou, Zhi Wang, Wu Zhou, and Xuxian Jiang. Hey, You, Get Off of My Market: Detecting Malicious Apps in Official and Alternative Android Markets. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2012.

11 Appendix

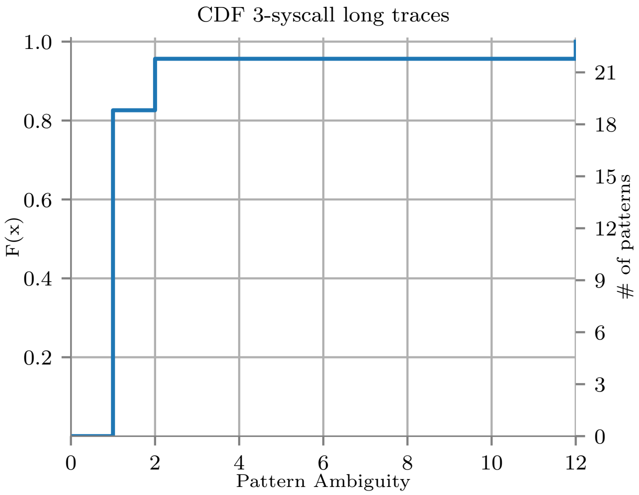

Figure 6: Pattern Ambiguity for 3-syscall long patterns

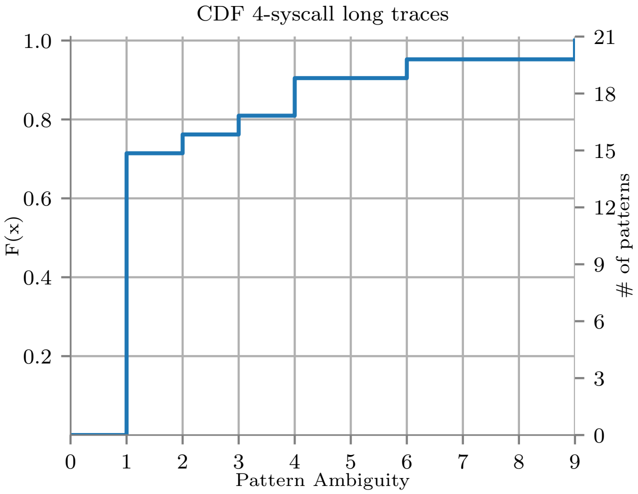

Figure 7: Pattern Ambiguity for 4-syscall long patterns

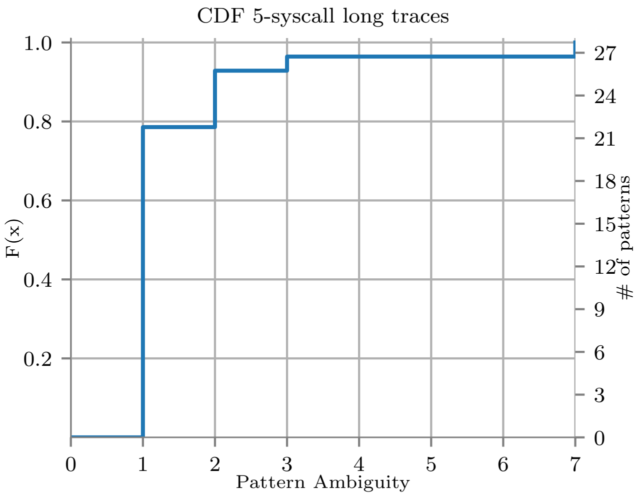

Figure 8: Pattern Ambiguity for 5-syscall long patterns

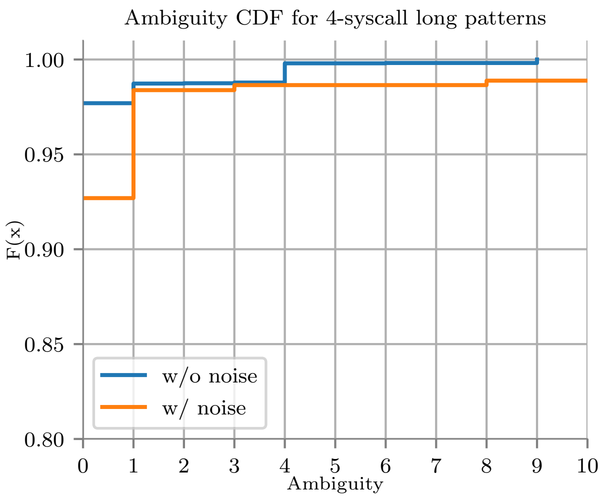

Figure 10: Total Pattern Ambiguity Comparison for 4-syscall long patterns

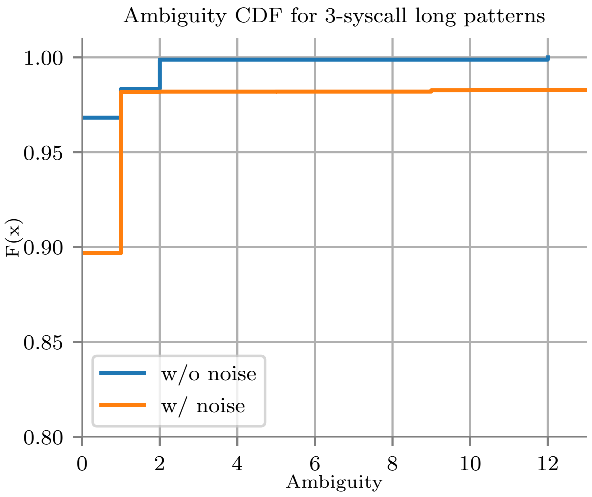

Figure 9: Total Pattern Ambiguity Comparison for 3-syscall long patterns

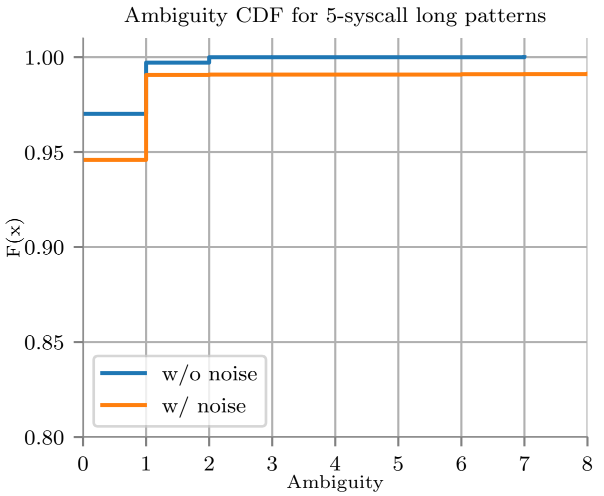

Figure 11: Total Pattern Ambiguity Comparison for 5-syscall long patterns