Malbehave: Classifying Malware by Observed Behavior

CS191W Project Report

Connor Gilbert, Bryce Cronkite-Ratcliff, Jason Franklin

Abstract—**Existing methods used to detect and characterize malicious software (malware) are highly labor-intensive and surprisingly easy to evade. The research community has produced a number of systems that attempt to improve these processes using behavioral characteristics. Here, we present a new system that recognizes and classifies Android malware based on observed program behavior. Our system learns the behaviors that define various classes of malware from program execution traces, producing recognizers that can be understood and usefully employed by human analysts. These recognizers can characterize malware in more detail than other systems, which mostly discriminate only between malicious and benign programs. Using knowledge of the Android platform, we generalize our recognizers to achieve improved robustness against code obfuscation techniques, especially code replacement. We present initial results showing our system's promise, and outline a number of areas for future work to improve its success.**

I. Introduction

Current malware analysis products rely on signatures that are easy to evade—detection can be circumvented using simple changes in program code. These signatures are generated using a highly labor-intensive, slow, expensive process that relies on continuous monitoring by skilled analysts. Although desktop security products have likely improved since troubling findings years ago about their vulnerability to simple evasive techniques [1], security researchers have very recently defeated popular Android security products using even the most basic modifications, such as changing an application's name or repackaging its components in a different order [2].

Once a software sample is identified as malware, often it is also useful to determine *in what way* it is malicious; this process, too, currently relies on skilled human analysis. It results in reports which are mainly intended to be read by humans, not machines. However, with these reports security vendors, researchers, and malware analysis companies have effectively produced a large labeled training set of malicious software that we can use to train a classifier to find those telltale actions.

Smart automation could improve malware detection effectiveness and allow skilled analysts to move beyond repetitive diagnosis and concentrate on finding new, pressing threats to computer security. We aim to automate the classification of malicious Android applications based on application behavior, using the intuition that software, in order to belong to a certain class of malicious applications, must do certain critical behaviors. For example, a premium-SMS-fraud application might often interact in certain ways with the SMS subsystem; a contact-stealing application might complete certain calls into both the contacts subsystem and the Internet subsystem; or a botnet-style application might carry out idiosyncratic behavior that indicates the existence of a covert communications or control channel.

Our approach is unique in the literature. Our aim is not only to automatically identify malicious behavior, but also to classify samples more specifically than simply as "malicious" or "benign". Our projected use is to augment human analysis rather than simply create another "box" to deploy. We exploit Android system properties to more accurately achieve robustness against code replacement. And, our approach specifically envisions modular recalibration of the detection system as threats change.

We first place our work in context in Section II. We describe our system and its implementation in Section III. We evaluate its performance in Section IV, outline further work in Section V and conclude in Section VI.

II. Related Work

The detection of malicious software is a topic of significant research interest. Fundamentally, the problem of behavioral malware detection is the problem of bridging the *semantic gap* between what a program

THIS IS A DRAFT REPORT. Please contact the authors if you wish to cite this work.

The authors are with the Computer Science Department at Stanford University, Stanford, CA 94305.

The first author was responsible for writing this report for CS191W under the direction of Prof. John Mitchell.

Email: connorg@cs.stanford.edu

does that is malicious and how that behavior can be observed. This concept is explored in the literature, *e.g.,* in [3] and later in [4]. In our description of existing work, we focus on each of a few fundamental tasks in malware detection: finding already-known malicious behaviors; discovering what behavior is malicious; and finding malware specifically on the Android platform.

Most recent approaches rely on system-call dependency graphs, following the intuition of [5] that system calls, since they are the only way to cause many damaging or security-relevant effects, are an effective way to detect malicious activity. We use a similar approach, but can provide richer logging because of Android's design features, including a richer permissions system, a recognizable set of sensitive data and metered resources, and a standard way to profile application behavior in great detail.

A. *Finding Known Malicious Behaviors*

Some older approaches analyze program text instead of execution traces. Bergeron et al. use a static analysis technique based on control-flow and data-flow graphs, though they provide no analysis of its effectiveness [6]. Their system requires security policies to be specified manually using "security automata", in which transitions are identified by system calls. The control-flow graph is filtered to include only calls to "critical APIs". Our approach similarly allows filtering, but provides a reproducible method to do so and evaluates its effectiveness.

Responding to the finding that commercial malware detectors performed poorly against basic obfuscation techniques [1], Christodorescu et al. present a "semantics-aware" detection scheme in [7]. The system is effective against certain obfuscations, using decision procedures and heuristics to establish whether sections of code are irrelevant to a program's effects and can thus be ignored in analysis. This system relies on hand-specified templates of malicious behavior, expressed in an intermediate semantic representation to achieve greater robustness against obfuscation. With similar goals and results, Kinder at al. introduce a new temporal logic in [8] and use model checking to verify that malicious behaviors, represented by hand-specified logical formulas, are absent from programs under consideration. Both techniques rely on matching instruction sequences rather than system calls. It is unclear whether this style of detection is still in use.

The use of static analysis has fallen out of favor, likely due in part to the theoretical difficulty and practical challenges of using static analysis tools against increasingly obfuscated code [9]. Static analysis efforts must also deal with encrypted or "packed" code—dynamic analysis tools can simply wait for such code to be deobfuscated and executed, then observe its effects.

Martignoni et al. attempt to address the semantic gap by constructing layered *behavior graphs* that specify events that, when observed together in an program execution trace, indicate that a specific meaningful behavior has occurred [4]. The system they present relies, again, on hand-specified descriptions of malicious behavior, but demonstrates the utility of "widening" detection graphs to include the multiple ways a specific action might be carried out, and therefore might be observed by a detector.

B. *Deriving Malicious Behavior Specifications*

Deriving specifications of malicious behavior—automatically discovering what, in fact, makes a program malicious—is a natural next step. More recent work, therefore, relies less on hand-specified templates of malicious behavior, and instead attempts to derive specifications without any *a priori* definition of malice.

Babić, Reynaud, and Song [10] introduce a technique similar to ours, which uses tree automata to classify software based on system call dependency graphs. Although our system currently uses grammar inference on strings of system calls, these techniques are similar.

Palahan et al. [11] present techniques to identify critical pieces of system call dependency graphs, toward the goal of recognizing high-confidence malicious behavior patterns in sets containing malware and goodware (software that is not malicious).

HOLMES, presented in [12], builds on previous work by many of its authors [13]. The system is comprised of a complex series of processing and inference steps that produce behavioral specifications for malware. While the system achieves good detection results, it relies on a number of unclear heuristic techniques, including curated lists of "security labels" derived from reading documentation.

Our desire to design a modular, tunable system is driven by results including [14], which found that simple, reasonable intuitions can actually lead malware detection researchers to make non-optimal design decisions.

C. *Android Malware Analysis*

While most of the the literature on malware analysis primarily concerns Windows or Linux malware, some recent work focuses specifically on Android.

Zhou and Jiang present a corpus of Android malware and provide useful analysis of its contents in [15]. Their results, based on a highly manual collection, categorization, and characterization process, are the basis of our current malware corpus.

Au et al. present the PScout tool and analysis of the Android permissions system in [16]. PScout uses static analysis to automatically annotate system methods with the permissions with which they are protected.

A number of systems attempt to identify malicious applications on Android. These mainly rely on system call interposition, similar to desktop efforts, but appear to base detection mostly on feature vectors for detection [17]–[19]. Such approaches, based on vectors of method call frequencies, are robust against reordering but can be evaded by inserting irrelevant or ineffectual calls.

In addition to systems that attempt to find actual malicious behavior, some tools analyze security configurations to identify potential security flaws. Such systems, for instance, detect when an application violates the principle of least privilege by requesting more privileges than are actually necessary for proper operation [20], or when multiple applications together request a dangerous combination of privileges [21].

D. Limitations

Although research efforts have narrowed the semantic gap in various ways, limitations remain. Recent work has demonstrated the ability of Android malware to detect when it is running in an emulator [22], [23]. The malware can infer that execution in a virtual environment means it is being analyzed for security purposes and simply fail to trigger its malicious behavior to avoid detection.

Emulators may also not provide a sufficiently faithful illusion of the “real world”, preventing certain malware from exhibiting its malice. For instance, a piece of malware might rely on network connectivity to specific websites to obtain commands, or might need to download additional code from a server. The research community has introduced various methods that achieve greater code coverage in spite of these limitations, either by providing richer emulation [12] or by detecting some of these dependencies and bypassing them [24].

These issues are beyond the scope of our work. Although these problems might prevent us from obtaining program traces that include malicious behavior, input generation and adversarial evasion are both already inherent problems for detection approaches that are based on dynamic analysis. Improvements in emulators or user-interaction simulators would simply provide better input to the rest of our process.

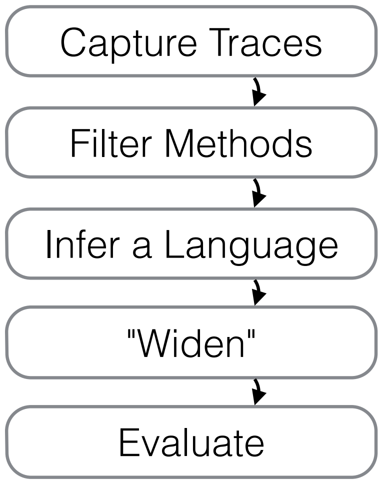


Fig. 1. A basic outline of our system’s design.

III. System Design

We use method calls to measure the behavior of each application under consideration. This is conceptually similar to using system calls (syscalls) to ascertain program behavior, a common technique in the literature [4], [5], [10]–[13]. We trace the program’s execution, filter the output, and use grammar inference to create a recognizer for each class of programs. We then transform the recognizer to make it more robust to adversarial program obfuscation, and, last, use the recognizer’s output. A simple diagram of this process is in Figure 1.

A. Capturing Program Traces

Our system is implemented using the standard Android profiler, which captures all method calls on all threads started by the application under consideration.

We inject simulated user behavior using the Android *Monkey* tool [25], which randomly generates user-interface events such as button presses and screen touches and gestures. Noting the finding of [15] that some malicious behavior is only triggered by certain external events, we use the Android Debug Bridge [26] to simulate external events including the reception of a telephone call, the reception of an SMS text message, and a change in the hardware’s power state.

The data returned by the profiler is extremely detailed, producing multiple megabytes of logs for even short executions. To concentrate on relevant behavior and more closely approximate syscall analysis, we filter the records using various techniques.

An example of a relevant trace output is the record of a call to an SMS-sending function shown in Figure 2.

Records of this type indicate the thread on which the activity was recorded (in Figure 2, thread 1); whether the

```
1 ent 531644
.....android/telephony/gsm/SmsManager.sendTextMessage
(Ljava/lang/String;Ljava/lang/String;
Ljava/lang/String;Landroid/app/PendingIntent;
Landroid/app/PendingIntent;)V
```

Fig. 2. An example of output from the trace.

call is <u>ent</u>ering or e<u>xit</u>ing (entering); the time in μsecs (531644); the stack depth (5, one level per '.'); the class and method name; and the method signature.[1]

We define various methods of filtering the records outputted by the profiler. Our implementation allows the following options:

- **Permissions:** Include any method that is protected by an Android permission, optionally outputting that permission. We obtain these annotations from the PScout project [16].
- **Data-flow:** Include any method that is either a possible source or a possible sink of sensitive data, optionally outputting that data-flow annotation. We obtain these annotations from the STAMP project at Stanford [27].
- **Android APIs:** Include any method, regardless of annotation, that is from an internal Android class.

By default, our implementation includes only those methods that bear either a permissions or data-flow annotation, since these are the behaviors most likely to involve private information or the use of a metered or billed resource. It also filters a small number of labels that are meaningful in other contexts but not for malicious behavior detection. Without an *a priori* definition of malice, we cannot make further judgments at this stage about whether a trace is malicious; we must leave this this task to the inference process.

Our implementation merges the activities of separate threads together in real-time chronological order of execution, but can be configured to treat each thread's method calls as separate traces. The output of this step is a series of filtered method calls which are then fed to the grammar inference step.

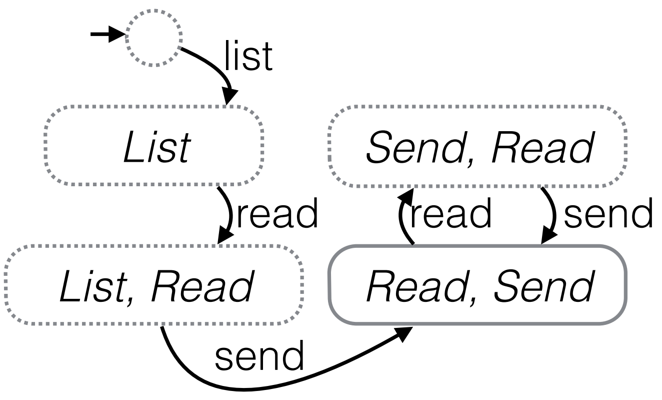
B. Creating a Recognizer

Once we have obtained program traces, we use standard grammar inference tools to create a recognizer. We currently use the *K-Testable* algorithm as implemented in the open-source *gitoolbox* project [28]. This algorithm infers a k-testable language [29], with individual system

[1]The method signature is expressed in the format of the Java specification, part 4.3.2. http://docs.oracle.com/javase/specs/jvms/se7/html/jvms-4.html#jvms-4.3.

```
*Trace 1*              *Trace 3*
ReadDirectoryList()    ReadDirectoryList()
NetworkRead()          NetworkRead()
NetworkSend()          NetworkSend()
                       NetworkRead()
*Trace 2*              NetworkSend()
ReadDirectoryList()    NetworkRead()
NetworkRead()          NetworkSend()
NetworkSend()
NetworkRead()
NetworkSend()
```

(a)



(b)

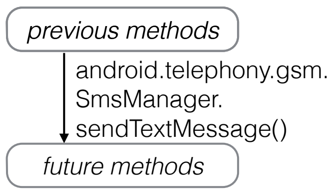
Fig. 3. An example of *K-Testable* inference using 3(a) as input traces. 3(b) is the DFA inferred using $k = 3$. The accept state is solid. Transitions are labeled with the event that would cause them to be taken. Inside each box is the prefix "remembered" by the DFA.

calls as its alphabet. The language is recognized by a DFA D that is returned by the inference algorithm; D accepts all traces on which it is trained, plus those included due to the algorithm's generalization based on the parameter k, which limits the amount of "memory" available to the detector to a sliding window of size k. While limiting the detector's window size might seem to reduce effectiveness, results from the literature indicate that small windows can achieve good detection results [5], in some cases better than larger windows [14]. Setting k to be longer than any existing trace demonstrates the potential for decreased performance with a larger window; with such a large k, the learned language would include only the training examples and no other strings.

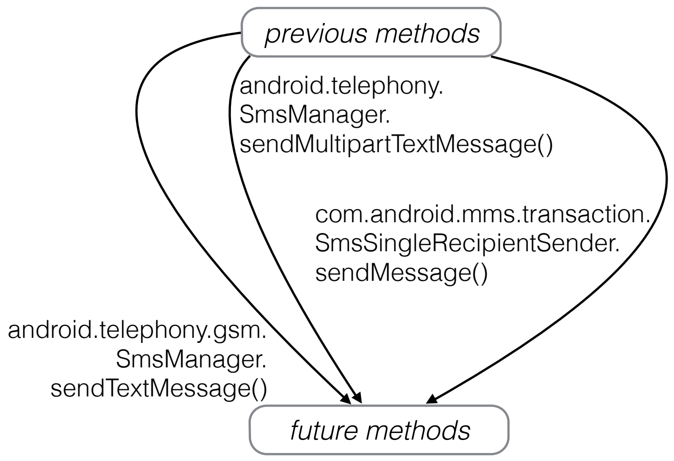
We illustrate the inference process with a simple example in Figure 3. We produced the DFA in 3(b) using *gitoolbox* from input directly transformed from 3(a). This example is simplified for space and readability; Figure 2 contains an example of the format of the name, class, and signature of a method as it would actually appear in a trace. This example might correspond to a hypothetical piece of malware that is receiving commands about which files to copy to an attacker's server. The learned DFA "remembers" $k-1$ symbols, so, in this case with $k=3$, the DFA tracks the two previous calls. Note that the language of the DFA includes any number (at least one) of repetitions of the Read and Send calls after a List call; this allowance is an example of the algorithm's generalization effects. If we add a trace that only has the first List call (i.e., consisting only of `ReadDirectoryList()`), the DFA simply marks the "List" state as an accept state; however, without such an example, the DFA will not accept that string.

We "widen" the inferred DFA to make it more robust against code replacement. The DFA should still recognize the malicious behavior even if one method call is replaced with another that could retrieve the same data or have the same effect. This technique is implemented through the use of the permissions and data-flow annotations discussed in III-A; if a method is annotated, its name in the DFA is effectively replaced with its annotation. In this way, any new method that is annotated identically to the first method (indicating that the two have similar effects) will also be recognized. This effect is illustrated in Figure 4. If we had simply used the automaton shown in 4(a), we would miss any of the additional methods shown in 4(b), allowing malware authors to evade detection by replacing the original method with one of these similar methods. The collapsed representation, which replaces the method with its permission label, is shown in 4(c).

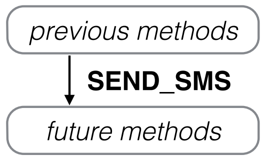
The DFAs produced by this method have, in our experience, been reasonably easy to understand. We anticipate that human analysts will be able to understand and refine the specifications derived from the inference process.



(a)



(b)



(c)

Fig. 4. An example of DFA widening. Transitions are labeled with the event that would cause them to be taken. The original DFA is modeled in 4(a). A conceptual example of "widening" is shown in 4(b), and a more concise representation is shown in 4(c).

IV. Evaluation

We carried out a number of experiments to evaluate the effectiveness of the techniques we have presented. All experiments reported here were conducted using the malware collected by Zhou and Jiang in the Malware Genome Project [15]. This dataset contains 1,260 samples from 49 separate families.

Our malware corpus is annotated with the class labels identified in the Malware Genome Project dataset [15]. These classifications are based on the following attributes: the method used to escalate privilege; the presence of a remote control channel; the unauthorized use of billable resources; and the theft of personal information. The number of example programs for each of these classes is shown in Table I; for details, see [15].

We attempted to evaluate the system with a set of goodware that we obtained by crawling a popular Android application store. However, we could not obtain adequate tracing data due to the applications' incompatibility with the old version of the Android OS (2.2.3) that we used due to the age of the malware corpus. Similarly, many of the malicious applications crashed immediately once launched or encountered other problems that interfered with tracing when executed.

We conducted two sets of experiments:

- **Malware Evolution:** A common measure of success for behavior-based malware is the ability of a signature generated with one version of malware to later detect successive versions of the same malware. In this set of experiments, we use traces from early *DroidKungFu* versions and measure detection rates against later versions.
- **Class Detection:** We used traces only from applications that belong to specific classes, as assigned in [15]. We measure detection against others from

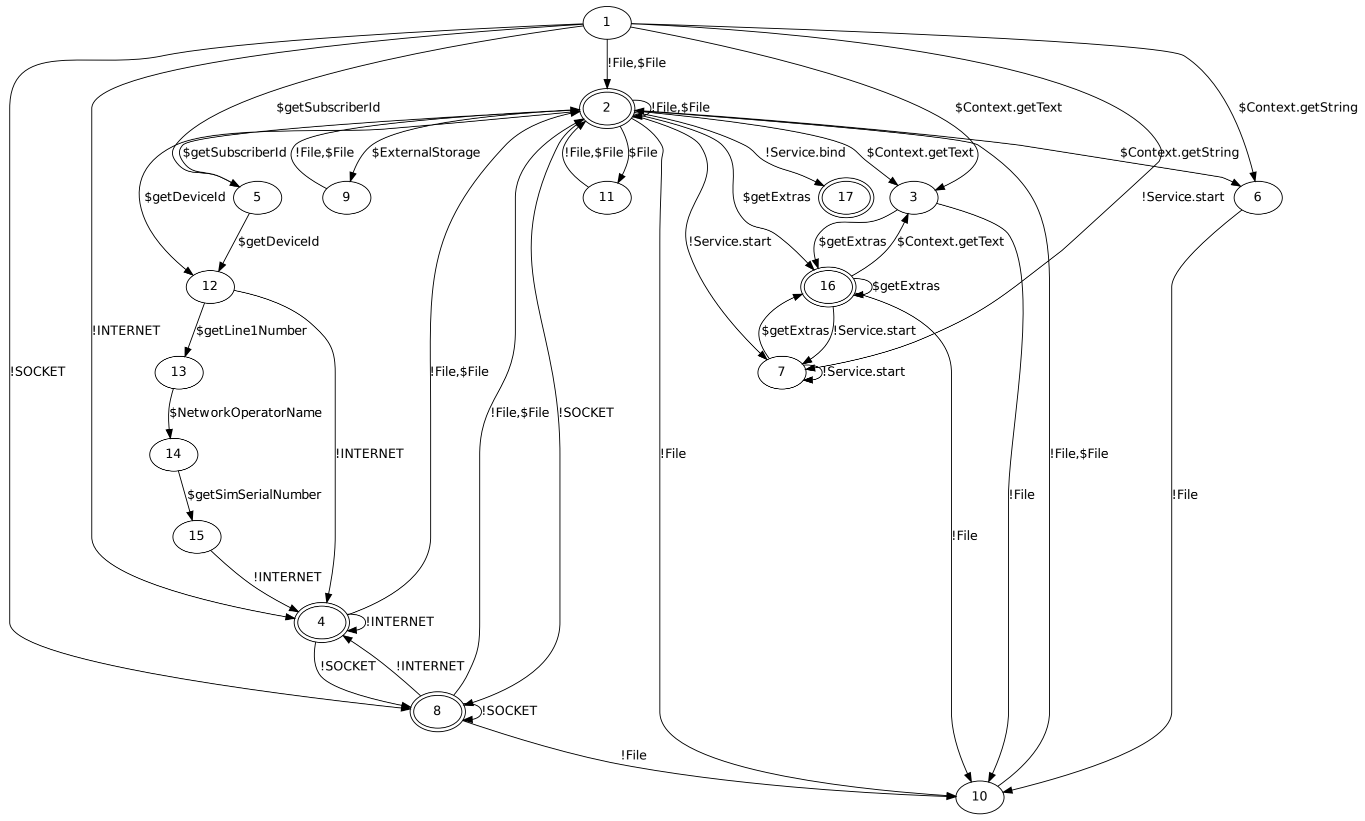


Fig. 5. The DFA learned by the k-testable inference algorithm using 16 traces from DroidKungFu1. The state numbers are not meaningful. The start state is labeled "1". Edges are labeled with data-flow annotations; while in state 1, for instance, encountering data-flow label `$Context.getString` in the trace would cause the DFA to move to state 6. Accept states, which indicate that a trace is a member of the learned language and thus in the class of DroidKungFu1-like malware, are marked with a double outline.

TABLE I
CLASS SIZES IN THE MALWARE GENOME PROJECT DATASET [15]

Class	**Families**	**Samples**
Privilege Escalation		
Exploid	6	389
RATC/Zimperlich	8	440
Ginger Break	1	4
Asroot	1	8
Encrypted	4	363
Remote Control		
Network	27	1171
SMS	1	1
Financial Charges		
Phone	4	246
SMS	28	571
Block SMS	17	315
Personal Information Stealing		
SMS	13	138
Phone Number	15	563
User Account	3	43

that set.

We use the experimental configurations in Table II.

TABLE II
EXPERIMENTAL CONFIGURATIONS

#	**Filters**	**Widening**
1	Permissions-protected only	Using permissions
2	Data-flow-annotated only	Using data-flow
3	Data-flow or permissions-protected	None

A. Malware Evolution

Our first test attempted to recognize successive generations of the *DroidKungFu* malware family. Details of the differences between each generation can be found in [15]. The behavioral detection process, even though it uses only the relatively simple input generation and grammar inference processes described above, detects a large number of the successive variants of DroidKungFu. In Figure 6, the first set of bars indicates the results obtained when training on DroidKungFu version 1 and testing against 2 and up, the second set indicates training on DroidKungFu versions 1 and 2 while testing against 3 and up, and so on. The inference process by design creates a detector that will recognize 100% of training

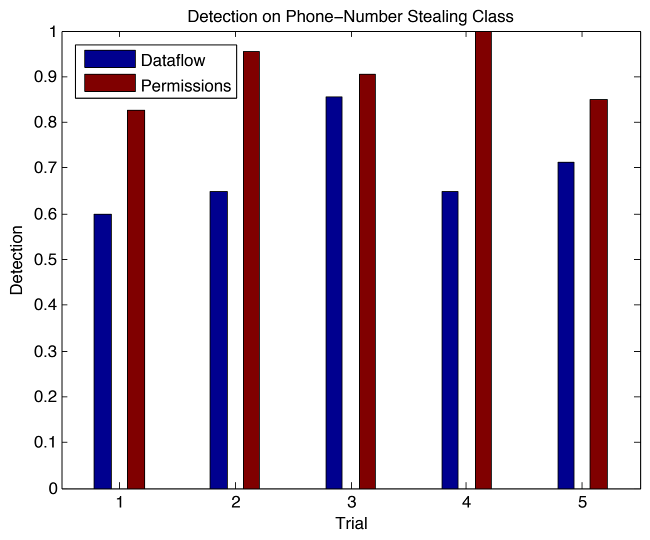


Fig. 8. Detection rates against a test set consisting of 20% of the malware that steals phone numbers, trained on the other 80% of this set of malware. The total size of this class is 113. Each trial was conducted using a separate partition into the training and test sets.

examples. The size of the test set decreases as we shift traces from successive variants of the DroidKungFu family from testing to training, but the detection rate increases.

The DFA learned by a Malware Evolution test is shown in Figure 5. While the DFA may appear somewhat large, it is notable that the recognizer for an entire class of malware contains only 17 states. Traces generally contain hundreds of thousands of method calls, producing files tens of megabytes long for each trace, which lasts less than a minute in the Android emulator. Filtering using data-flow or permissions labels significantly reduces the size of the traces, down to tens or hundreds of events per trace, and sometimes even down to zero if no permissions-protected or data-flow-relevant actions were performed. Inference took under a minute in all cases, and many times completed within a few seconds.

This experience also shows the value of the widening process. The grammars learned using method names (even filtered to include only "interesting" calls, defined as those with either a data-flow or permissions label), achieved worse detection results compared with those learned using widening processes. In addition, the inference process failed to complete when presented with the large vocabulary present in a language consisting of so many method names.

False-positive rates using malware of types other than DroidKungFu are plotted in Figure 7. Once an acceptable set of goodware can be obtained, we will generate more detailed false-positive statistics. DroidKungFu shares many behavioral characteristics with other malware families, so the performance against non-DroidKungFu malware illustrates how specifically the learned model fits the DroidKungFu family.

B. *Class Detection*

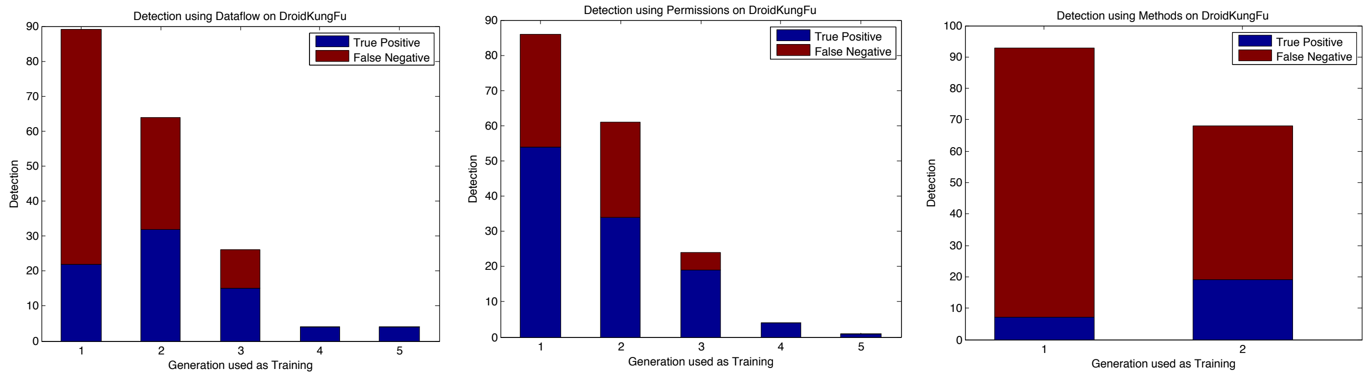
Our next experiment demonstrates our system's utility for the recognition of behavioral classes that are more specific than simply "malicious" versus "benign". We ran our system on the class of malware that steals phone numbers. We divided the set of malware into five trials with training and test sets that contained 80% and 20% of the traces, respectively. The detection rates against the test sets are shown in Figure 8. (As noted before, the inference process creates a detector that will recognize 100% of training examples.) We computed false positive rates by training with the entire set of phone-number-stealing malware, then testing on the traces from other types of malware. Using dataflow, the detector mistakenly accepted 45% of the traces; with permissions, the detector mistakenly accepted 69% of the traces. In both cases, this rate is lower than the detection rates against true members of the class as shown in Figure 8, suggesting that a positive result provides useful information.

V. Further Work

This paper demonstrates the promise of our approach, but there are a number of ways that the preliminary design described here could be improved.

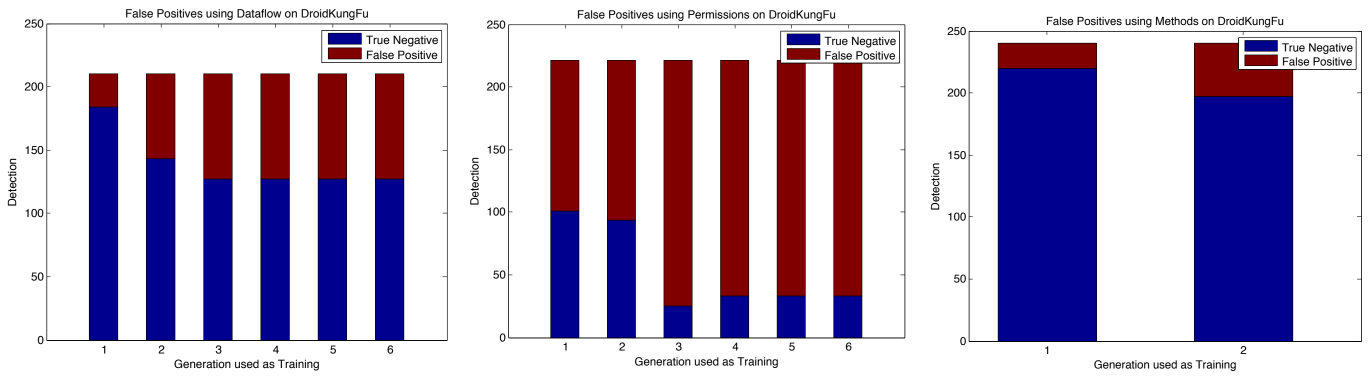
Our system currently employs Android's default *Monkey* tool to generate random input to applications under test. More sophisticated techniques have been shown to obtain significantly better code coverage [24], and we believe similar work would allow our detection system to identify malware that requires more complicated conditions to trigger malicious actions. In addition, newer testing tools like *Dynodroid* produce a larger variety of events [30], including the types of events that trigger many malware families in our malicious corpus. These tools may therefore provide more opportunities to trigger malicious behavior in the programs being tested and learn more tell-tale signs of malice.

To avoid false positives, we need a way to remove the behaviors that are exhibited by programs outside of the class we are attempting to describe (for instance, goodware, or malware of a different type). We plan to accomplish this goal by removing paths from the inferred DFA or using probabilistic process that would discount behaviors that also occur in goodware, perhaps in a way



(a) Data-Flow (b) Permissions (c) Methods

Fig. 6. Detection success using the DFA learned from the DroidKungFu family. The horizontal axis indicates which generation is used as the training set; generation 5 indicates 1-4 plus the 'DroidKungFuSapp' variation. The height of the bar is the total size of the test set, and the colors indicate success or failure of detection. The DFA for generation 1 data-flow is in Figure 5. The size of the training set overwhelmed the inference process in 6(c) after the first and second generations were included.



(a) Data-Flow (b) Permissions (c) Methods

Fig. 7. Detection results using the DFA learned from the DroidKungFu family, tested against malware not of the DroidKungFu family. The horizontal axis indicates which generation is used as the training set; generation 5 indicates 1-4 plus the 'DroidKungFuSapp' variation, and generation 6 also includes 'DroidKungFuUpdate'. The height of the bar is the total size of the test set, and the colors indicate success or failure of detection; a "true negative" means that the detector did not recognize the non-DroidKungFu malware sample. The size of the training set overwhelmed the inference process in 7(c) after the first and second generations were included.

similar to that used in the production of stochastic k-testable tree languages [31]. This work might also help isolate the most critical, tell-tale behaviors present in each class.

More generally, we note that most malware detection systems found in the literature concentrate on a single type of detection—for instance, as in our system, only on behavior. However, improved results might be obtained by combining indicators from different types of analysis, for instance by using behavioral analysis, permissions configuration, and static analysis techniques together.

Our system is built to allow modular experimentation, and we plan to explore the use of new techniques at each step of the process: behavioral monitoring, filtering and widening, inference, and analysis.

VI. Conclusions

We presented the design of a new system to detect and classify malware on the Android platform through behavioral analysis. Our system design drew on the insights of the research community's existing work on behavioral analysis and applied features unique to the Android system, including a rich permissions system and well-defined interface for access to sensitive subsystems, to improve robustness against common obfuscation techniques. Using our initial system implementation, we demonstrated that behavioral analysis is a promising method for Android malware detection and classification. We plan to explore improvements in various parts of our system to achieve improved results.

VII. Acknowledgements

We wish to thank Yajin Zhou and Xuxian Jiang for access to the Malware Genome Project, their categorized corpus of Android malware [15]; Kathy Wain Yee Au, Yi Fan Zhou, Zhen Huang, and David Lie for publishing annotations of permission-protected Android API methods generated with their PScout tool [16]; Hasan Ibne Akram, Colin de la Higuera, Huang Xiao, and Claudia Eckert for making gitoolbox publicly available [28]; and the STAMP team at Stanford for producing detailed dataflow annotations of the Android system [27].

References

[1] M. Christodorescu and S. Jha, "Testing malware detectors," in *Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA '04. New York, NY, USA: ACM, 2004, pp. 34–44. [Online]. Available: http://doi.acm.org/10.1145/1007512.1007518

[2] V. Rastogi, Y. Chen, and X. Jiang, "DroidChameleon: evaluating Android anti-malware against transformation attacks," in *Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security*, ser. ASIA CCS '13. New York, NY, USA: ACM, 2013, pp. 329–334. [Online]. Available: http://doi.acm.org/10.1145/2484313.2484355

[3] P. Chen and B. Noble, "When virtual is better than real [operating system relocation to virtual machines]," in *Hot Topics in Operating Systems, 2001. Proceedings of the Eighth Workshop on*, May 2001, pp. 133–138.

[4] L. Martignoni, E. Stinson, M. Fredrikson, S. Jha, and J. C. Mitchell, "A layered architecture for detecting malicious behaviors," in *Proceedings of the 11th International Symposium on Recent Advances in Intrusion Detection*, ser. RAID '08. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 78–97. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-87403-4_5

[5] S. Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff, "A sense of self for unix processes," in *Proceedings of the 1996 IEEE Symposium on Security and Privacy*, ser. SP '96. Washington, DC, USA: IEEE Computer Society, 1996, pp. 120–128. [Online]. Available: http://dl.acm.org/citation.cfm?id=525080.884258

[6] J. Bergeron, M. Debbabi, J. Desharnais, M. M. Erhioui, Y. Lavoie, and N. Tawbi, "Static detection of malicious code in executable programs," *Int. J. of Req. Eng*, 2001.

[7] M. Christodorescu, S. Jha, S. A. Seshia, D. Song, and R. E. Bryant, "Semantics-aware malware detection," in *Proceedings of the 2005 IEEE Symposium on Security and Privacy*, ser. SP '05. Washington, DC, USA: IEEE Computer Society, 2005, pp. 32–46. [Online]. Available: http://dx.doi.org/10.1109/SP.2005.20

[8] J. Kinder, S. Katzenbeisser, C. Schallhart, and H. Veith, "Detecting malicious code by model checking," in *Proceedings of the Second International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, ser. DIMVA'05. Berlin, Heidelberg: Springer-Verlag, 2005, pp. 174–187. [Online]. Available: http://dx.doi.org/10.1007/11506881_11

[9] A. Moser, C. Kruegel, and E. Kirda, "Limits of static analysis for malware detection," in *Computer Security Applications Conference, 2007. ACSAC 2007. Twenty-Third Annual*, Dec 2007, pp. 421–430.

[10] D. Babić, D. Reynaud, and D. Song, "Recognizing malicious software behaviors with tree automata inference," *Form. Methods Syst. Des.*, vol. 41, no. 1, pp. 107–128, Aug. 2012. [Online]. Available: http://dx.doi.org/10.1007/s10703-012-0149-1

[11] S. Palahan, D. Babić, S. Chaudhuri, and D. Kifer, "Extraction of statistically significant malware behaviors," in *Proceedings of the 29th Annual Computer Security Applications Conference*, ser. ACSAC '13. New York, NY, USA: ACM, 2013, pp. 69–78. [Online]. Available: http://doi.acm.org/10.1145/2523649.2523659

[12] M. Fredrikson, S. Jha, M. Christodorescu, R. Sailer, and X. Yan, "Synthesizing near-optimal malware specifications from suspicious behaviors," in *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, ser. SP '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 45–60. [Online]. Available: http://dx.doi.org/10.1109/SP.2010.11

[13] M. Christodorescu, S. Jha, and C. Kruegel, "Mining specifications of malicious behavior," in *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ser. ESEC-FSE '07. New York, NY, USA: ACM, 2007, pp. 5–14. [Online]. Available: http://doi.acm.org/10.1145/1287624.1287628

[14] D. Canali, A. Lanzi, D. Balzarotti, C. Kruegel, M. Christodorescu, and E. Kirda, "A quantitative study of accuracy in system call-based malware detection," in *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, ser. ISSTA 2012. New York, NY, USA: ACM, 2012, pp. 122–132. [Online]. Available: http://doi.acm.org/10.1145/2338965.2336768

[15] Y. Zhou and X. Jiang, "Dissecting Android malware: Characterization and evolution," in *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, ser. SP '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 95–109. [Online]. Available: http://dx.doi.org/10.1109/SP.2012.16

[16] K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie, "PScout: Analyzing the Android permission specification," in *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, ser. CCS '12. New York, NY, USA: ACM, 2012, pp. 217–228. [Online]. Available: http://doi.acm.org/10.1145/2382196.2382222

[17] I. Burguera, U. Zurutuza, and S. Nadjm-Tehrani, "Crowdroid: Behavior-based malware detection system for Android," in *Proceedings of the 1st ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, ser. SPSM '11. New York, NY, USA: ACM, 2011, pp. 15–26. [Online]. Available: http://doi.acm.org/10.1145/2046614.2046619

[18] G. Dini, F. Martinelli, A. Saracino, and D. Sgandurra, "MADAM: A multi-level anomaly detector for Android malware," in *Proceedings of the 6th International Conference on Mathematical Methods, Models and Architectures for Computer Network Security: Computer Network Security*, ser. MMM-ACNS'12. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 240–253. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-33704-8_21

[19] J.-w. Jang, J. Yun, J. Woo, and H. K. Kim, "Andro-profiler: Anti-malware system based on behavior profiling of mobile malware," in *Proceedings of the Companion Publication of the 23rd International Conference on World Wide Web Companion*, ser. WWW Companion '14. Republic and Canton of Geneva, Switzerland: International World Wide Web Conferences Steering Committee, 2014, pp. 737–738. [Online]. Available: http://dx.doi.org/10.1145/2567948.2579366

[20] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner, "Android permissions demystified," in *Proceedings of the 18th ACM Conference on Computer and Communications*

Security, ser. CCS '11. New York, NY, USA: ACM, 2011, pp. 627–638. [Online]. Available: http://doi.acm.org/10.1145/2046707.2046779

[21] A. P. Felt, H. J. Wang, A. Moshchuk, S. Hanna, and E. Chin, "Permission re-delegation: Attacks and defenses," in *Proceedings of the 20th USENIX Conference on Security*, ser. SEC'11. Berkeley, CA, USA: USENIX Association, 2011, pp. 22–22. [Online]. Available: http://dl.acm.org/citation.cfm?id=2028067.2028089

[22] T. Petsas, G. Voyatzis, E. Athanasopoulos, M. Polychronakis, and S. Ioannidis, "Rage against the virtual machine: Hindering dynamic analysis of Android malware," in *Proceedings of the Seventh European Workshop on System Security*, ser. EuroSec '14. New York, NY, USA: ACM, 2014, pp. 5:1–5:6. [Online]. Available: http://doi.acm.org/10.1145/2592791.2592796

[23] J. Oberheide and C. Miller, "Dissecting the Android Bouncer," in *SummerCon '12*, June 2012. [Online]. Available: https://jon.oberheide.org/files/summercon12-bouncer.pdf

[24] 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. [Online]. Available: http://dx.doi.org/10.1109/SP.2007.17

[25] Android, "UI/Application exerciser monkey." [Online]. Available: http://developer.android.com/tools/help/monkey.html

[26] ——, "Android debug bridge." [Online]. Available: http://developer.android.com/tools/help/adb.html

[27] "STatic Analysis of Mobile Programs (STAMP)." [Online]. Available: https://sites.google.com/site/stampwebsite/home

[28] H. I. Akram, C. de la Higuera, H. Xiao, and C. Eckert, "Grammatical inference algorithms in matlab," in *Proceedings of the 10th International Colloquium on Grammatical Inference*, ser. ICGI 2010. Springer-Verlag, 2010.

[29] P. García and E. Vidal, "Inference of k-testable languages in the strict sense and application to syntactic pattern recognition," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 12, no. 9, pp. 920–925, Sep. 1990. [Online]. Available: http://dx.doi.org/10.1109/34.57687

[30] A. Machiry, R. Tahiliani, and M. Naik, "Dynodroid: An input generation system for Android apps," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2013. New York, NY, USA: ACM, 2013, pp. 224–234. [Online]. Available: http://doi.acm.org/10.1145/2491411.2491450

[31] J. Rico-Juan, J. Calera-Rubio, and R. Carrasco, "Stochastic k-testable tree languages and applications," in *Grammatical Inference: Algorithms and Applications*, ser. Lecture Notes in Computer Science, P. Adriaans, H. Fernau, and M. Zaanen, Eds. Springer Berlin Heidelberg, 2002, vol. 2484, pp. 199–212. [Online]. Available: http://dx.doi.org/10.1007/3-540-45790-9_16