Development of Mobile Malware Detection Technique Based on Semantics-Aware and Signature

Helanfeng Sun
Case Western Reserve University
hxs593@case.edu

Tan Li
Case Western Reserve University
txl555@case.edu

ABSTRACT

We do our survey on six papers from 2005 to 2014 in area of signature-based malware detection, semantics-based malware detection, and how they can be combined together to obtain a much better tool of Android platform. These tools include IDA Pro the disassembly tool, Hancock from Symantec, Kiri, TaintDroid and Apposcopy. As time goes, accuracy increase and development of technology are witnessed.

1. INTRODUCTION

Malware detection technique had been studied for a long time before smart phones were invented. To evade detection, hackers use obfuscation skills to deceive traditional pattern-matching virus scanners. The primary deficiency of pattern-matching detection technique is that it is syntactic-based and have no relation with semantic information inside the program. Semantics-aware malware detection algorithm is resilient to common obfuscations used by hackers. Scanning files for signature is also a proven technology with weakness. Facing the explosion of malware program, researchers also have to deal with the exponential growth of signature database. Signature-based technique can minimize the false positive rate of malware matching. In smart phone era, number of malwares increases rapidly on Android, which is the most commonly used system in the world. So this survey keeps eye on combining skill of semantics analysis and signature-based detection to detect Android malware.

2. BACKGROUND

2.1.1 Android Components

Activity: An activity is a single, focused thing that the user can do. Almost all activities interact with the user, so the Activity class takes care of creating a window for you in which you can place your UI with setContentView(View). While activities are often presented to the user as full-screen windows, they can also be used in other ways: as floating windows or embedded inside of another activity.

Service: A Service is an application component representing either an application's desire to perform a longer-running operation while not interacting with the user or to supply functionality for other applications to use.

Broadcast: Android apps can send or receive broadcast messages from the Android system and other Android apps, similar to the publish-subscribe design pattern. These broadcasts are sent when an event of interest occurs.

ContentProvider: Content providers are one of the primary building blocks of Android applications, providing content to applications. They encapsulate data and provide it to applications through the single ContentResolver interface. A content provider is only required if you need to share data between multiple applications. For example, the contacts data is used by multiple applications and must be stored in a content provider. If you don't need to share data amongst multiple applications you can use a database directly via SQLiteDatabase.

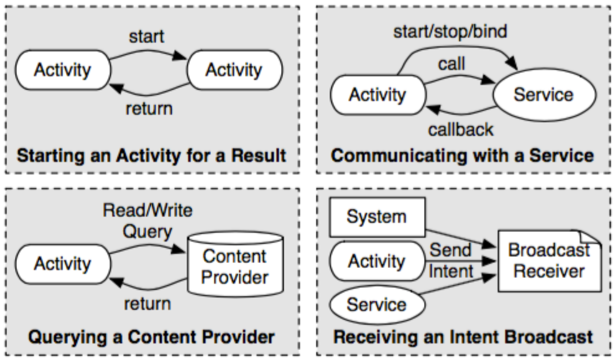


Figure1. Android Components Relationships

2.1.2 Android Applications' Life Cycle

As a user navigates through, out of, and back to your app, the Activity instances in your app transition through different states in their lifecycle. The Activity class provides a number of callbacks that allow the activity to know that a state has changed: that the system is creating, stopping, or resuming an activity, or destroying the process in which the activity resides.

Within the lifecycle callback methods, you can declare how your activity behaves when the user leaves and re-enters the activity. For example, if you're building a streaming video player, you might pause the video and terminate the network connection when the user switches to another app. When the user returns, you can reconnect to the network and allow the user to resume the video from the same spot. In other words, each callback allows you to perform specific work that's appropriate to a given change of state. Doing the right work at the right time and handling transitions properly make your app more robust and performant.

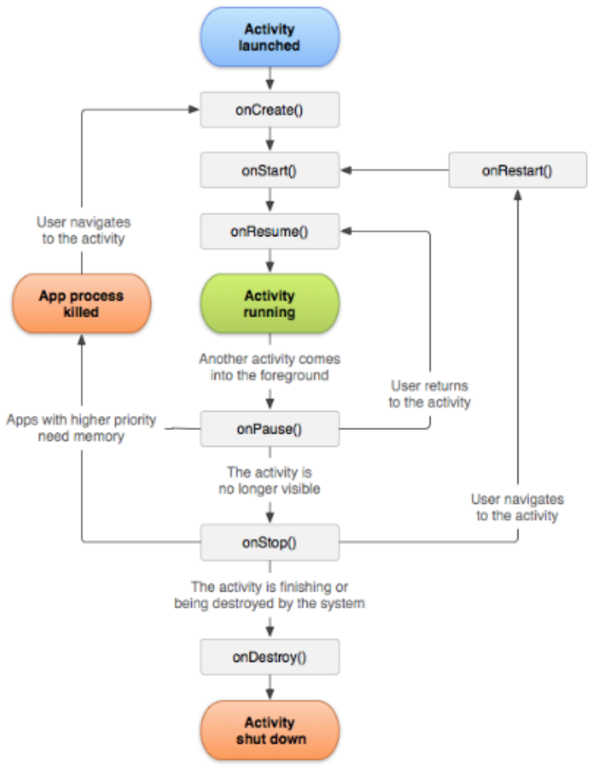

Activity launched
onCreate()
onStart()
onRestart()
User navigates to the activity
onResume()
App process killed
Activity running
Another activity comes into the foreground
User returns to the activity
Apps with higher priority need memory
onPause()
The activity is no longer visible
User navigates to the activity
onStop()
The activity is finishing or being destroyed by the system
onDestroy()
Activity shut down

Figure2. Android Activity Life Cycle

3. STUDY

3.1 Semantic-Aware Malware Detection

Malware instance is a program with malicious intents and signature-based virus scanner is an example of malware detector. Though, this pattern-matching detection is not useful when they are faced with obfuscation. Malware writers would use polymorphism and metamorphism while writing virus, which means a virus can morph itself by decrypting during execution but encrypting its malicious payload when they are not activated, or they might change their code during replication. Such technique can easily defeat the pattern-matching detection solution. In 2005, team of University of Wisconsin, Madison and Carnegie Mellon University find a malware detection algorithm that uses semantics of instructions of a malware, which will minor obfuscation and variations.

3.1.1 Formal Semantic Analysis

Behavior of a program can be expressed in control-flow graph like Figure1.

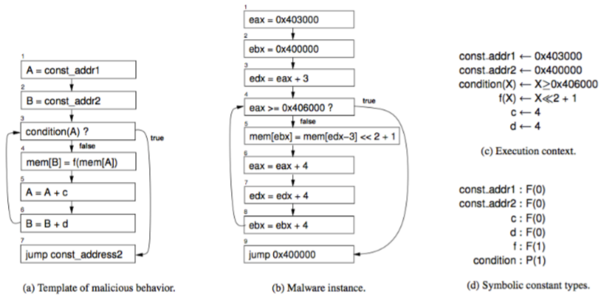


Figure 1. Malware instance (b) satisfies the template (a) according to our semantics.

Figure 1(a) describes a malicious behavior that is a simplified version of a decryption loop found in polymorphic worms. Consider the instruction sequence shown in Figure 1(b). Assume the symbolic constants in the template are assigned values shown in Figure 1(c). This shows the malicious behavior is not affected while the malware transform, like some detail like register renaming and changing starting addresses of memory blocks.

While doing the formal semantics analysis, the author of this semantics-based analysis two definitions with two conditions and one theorem. To meet the need of simplify the semantics matching, a modified condition is also generated. These definitions and less than the year 2014 paper about Apposcopy because researchers of that paper works with Android system which means the control flow graph is not defined like Figure1 but taking Android component as node and call path as edge. The difficulty of Apposcopy is based on its signature generate algorithm. Thus, definitions in Apposcopy theory is much more than this semantics detection.

They still have similarity that both of these two papers create family identifier for these malwares. Malware family tag ia based on certain malicious behaviors, with which we can easily figure out new malwares' action.

3.1.2 Architechture of Semantics-aware matching algorithm

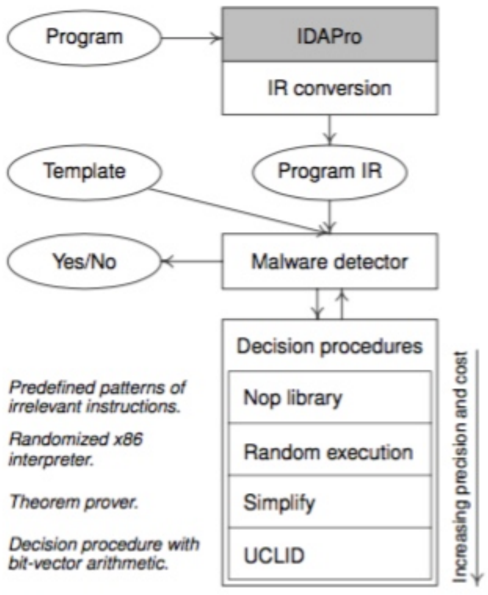

Program
IDAPro
IR conversion
Template
Program IR
Yes/No
Malware detector
Decision procedures
Predefined patterns of irrelevant instructions.
Nop library
Randomized x86 interpreter.
Random execution
Theorem prover.
Simplify
Decision procedure with bit-vector arithmetic.
UCLID
Increasing precision and cost

Figure 2. The architecture of the malicious code detector. Gray boxes represent existing infrastructure.

The tool architecture is Figure2. This IDA Pro disassembler based tool start with the input of binary program. We first disassemble the binary program, construct control flow graphs, and produce an intermediate representation(IR). The IR is generated using a library of x86 instruction transformers, in such a way that the IR is architecture-independent. The IR can still contain library and system calls.

The rest of the toolkit takes advantage of the platform and architecture-independent IR. By providing appropriate front-ends that translate the program into the IR format, one can use this malware-detector tool to process programs for different platforms.

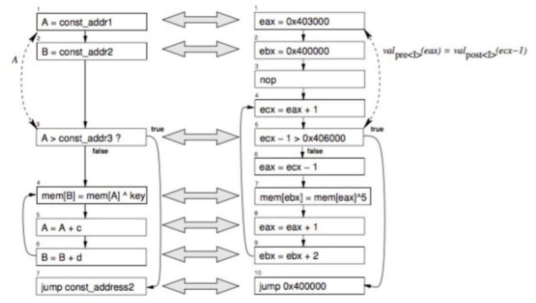


Figure 3. Example of program (on the right) satisfying a template (on the left) according to our algorithm $\mathcal{A}_{MD}$. Gray arrows connect program nodes to their template counterparts. The dashed arrow on the left marks one of the def-use relations that *does hold true* in the template, while the corresponding dashed arrow on the right marks the def-use relation that *must hold true* in the program.

Way to match teh malware with a template is that, first match the template nodes to program nodes and second preserve the def-use paths. Figure3 shows the implementation of formal detection approached of matching. The unification step addresses the first condition of algorithm AMD, by producing an assignment to variables in an instruction at a template node such that it matches a program node instruction. The matching algorithm takes these restrictions into account: (1) A variable in the template can be unified with any program expression, except for assignment expressions; (2) A symbolic constant in the template can only be uni- fied with a program constant; (3) An external function call in the template can only be unified with the same external function call in the program; (4) A predefined operator in the template can only be unified with the same operator in the program.

Unified nodes		*Bindings*

T1	P1	A ← eax
		const_addr1 ← 0x403000

T2	P2	B ← ebx
		const_addr2 ← 0x400000

T3	P5	A ← ecx - 1
		const_addr3 ← 0x406000

T4	P7	A ← eax
		B ← ebx

T5	P8	A ← eax
		increment1 ← 1

T6	P9	B ← ebx
		increment2 ← 2

T7	P10	const_addr2 ← 0x400000

Table 1. Bindings generated from the unification of template and program nodes in Figure 3. Notation Tn refers to node n of the template, and Pm refers to node m of the program.

Then Table1 can be generated by bindings generated from the unification of template and program nodes. By using preservation on def-use chains. The Amd Algorithm can handle obfuscation transformation like instruction reorderig, register renaming, garbage insertion but still have limitation in instruction replacing equivalent functionality and reorder memory accesses.

3.1.3 *Evaluations on semantics-based malware detection*

The research focus on three evaluations: (1) The template-based algorithm detects worms from the same family, as well as unrelated worms, using a single template. (2) No false positives were generated when running our malware detector on benign programs, illustrating the soundness of our algorithm in its current implementation. (3) The algorithm exhibits improved resilience to obfuscation when compared to commercial anti-virus tool McAfee VirusScan.

Malware family	*Template detection*		*Running time*	
	Decryption loop	*Mass-mailer*	*Avg.*	*Std. dev.*
Netsky	100%	100%	99.57 s	41.01 s
B[e]agle	100%	100%	56.41 s	40.72 s
Sober	100%	0%	100.12 s	45.00 s

Table 4. Malware detection using algorithm $\mathcal{A}_{MD}$ for 21 e-mail worm instances.

Table4 shows the evaluation on each malware instance and each decryption-loop template, which shows the detection on Netsky and B[e]agle worm achieves 100% and running times for the tool(in Table4) are satisfactory for a prototype and suggest that real-time performance can be achieved with an optimized implementation.

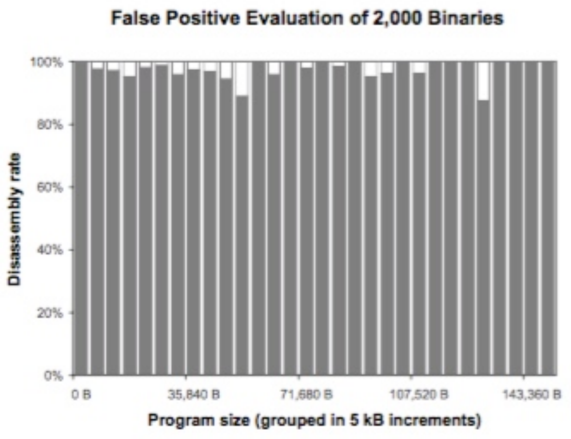
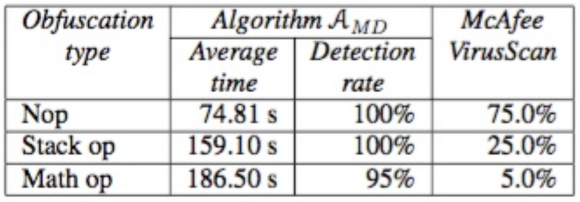


Figure 4. Evaluation of algorithm $\mathcal{A}_{MD}$ on a set of 2,000 benign Windows programs yielded no false positives. Gray bars indicate the percentage of programs that disassembled correctly and were detected as benign; white bars indicate the percentage of programs that failed to disassemble.

The disassembly rate is also high with only 2.22% failed to disassembly. The comparison with Macfee VirusScan shows semantics-based tool is much more powerful.



Obfuscation type	Algorithm $\mathcal{A}_{MD}$		McAfee VirusScan
	Average time	Detection rate	
Nop	74.81 s	100%	75.0%
Stack op	159.10 s	100%	25.0%
Math op	186.50 s	95%	5.0%

Table 5. Evaluation of algorithm $\mathcal{A}_{MD}$ on a set of obfuscated variants of B[e]agle.Y. For comparison, we include the detection rate of McAfee VirusScan.

3.2 Automatic Generation of String Signatures for Malware Detection

3.2.1 Introduction

Signature-based malware scanning is still the dominant approach to identifying malwares because of its extremely low false positive rate. In this paper, the author provide a way to generate string signature to detect malware. This tool Hancock can generate string signatures with a false positive rate below 0.1% by focusing on automatically generating string signatures with high-coverage.

3.2.2 Signature Candidate Selection

Hancock has two important features: (1) Scalable to very large goodware set; (2) Focusing on rare byte sequences. The basic algorithm model used in Hancock is a 5-gram Markov chain model. The model pruning of the model training set and memory requirement is optimized to keep the most valuable grams, given a fixed memory constraint. In the Model merging process, the available memory limits the size of the model that can be created. To go over it, Hancock uses several smaller models on subsets of the training data, prunes them, and get them merged.

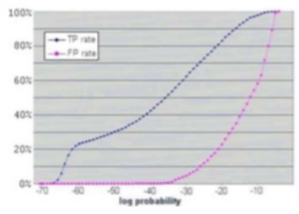


Fig. 1. Fractions of FP and TP sequences with probabilities below the X value

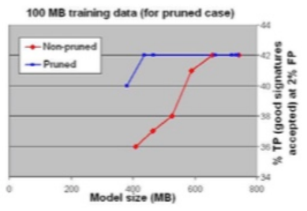


Fig. 2. TP rate comparison between models with varying pruning thresholds and varying training set sizes

The result of the signature selection process is shown in Figure1 and Figure2. As expected, true positive signatures have much lower probabilities than false positive signatures. A small number of false positive signatures have low probabilities. And the model does have discrimination ability, rejecting 99% of false positive signatures and accepting almost half of true positive signatures. Figure2 shows pruning can almost halve the goodware model size and offer the same true positive rate as the pruned model derived from the same training set.

Library function Recognation is based on IDA Pro's Fast Library Iden- tification and Recognition Technology(FLIRT). In contrast to FLIRT, Hancock aims to eliminate flase positive signatures. By modifying Universal FLIRT heuristic to library function referenced heuristic, the proposed technique disassembles a binary program, builds a call graph representation of the program, and marks any method that is called by a known library function. This process to mark functions repeats again and again until no new library function can be found.

This C++ reference technique has almost same theory with marking Android SDK in Apposcopy. Both of them focus on making lable in library. However, Apposcopy modify source and sink method in Android SDK manually, but Hancock mark library on learning model.

Code interestingness check is designed with unusual address offets, funcion call listening and math instructions. These operations are sometimes used in obfuscation.

3.2.3 Signature Candidate Filtering

On byte-level, Hancock cluster malwares into different families. Hancock also use signature position deviation and multiple common signatures to analysis whether to create a new family for malwares. Surrounding Context Count helps Hancock repeats the same expansion procedure if the expanded byte sequences reach limitation in size or number of distinct expanded byte sequences gets over a threshold.

On instruction level, Hancock can extract enclosing function in certain malware file of certain family, the rules are as follows: (1) Two enclosing functions are identical in byte sequences; (2) The instruction op-code sequences of the two enclosing functions are also identical. (3) After instruction sequence normalization, the instruction op-code sequences of the two enclosing functions are still identical.

3.2.4 Multi-Component String (MCS) Signature Generation

This string generation technique is based on single-component signature(SCS). Hancock chooses three to five components to generate signature in each MCS. The algorithm is following: Given a set of qualified component signature candidates, Hancock uses the following algorithm to arrive at the final subset of component signature candidates used to form MCSs, S2: (1) Compute for each component signature candidate in S1 its effective coverage value, which is a sum of weights associated with each file the component signature candidate covers. The weight of a covered file is equal to its coverage count, the number of candidates in S2 already covering it, except when the number of component signatures in S2 covering that file is larger than or equal to K, in which case the weight is set to zero.

(2) Move the component signature candidate with the highest effective cover- age value from S1 to S2, and increment the coverage count of each file the component signature candidate covers.

(3) If there are still malware files that are still uncovered or there exists at least one component signature in S1 whose effective coverage value is non-zero, go to Step 1; otherwise exit.

The above algorithm is modified from set covering problem. After S2 is determined, Hancock will keep test the signatures and get MCS with better false positive rate until threshold.

3.2.5 *Evaluations on generated signatures*

Table 6. Multi-Component Signature results

# components	Permitted component FPs	Coverage	# Signatures	# FPs
2	1	28.9%	76	7
2	0	23.3%	52	2
3	1	26.9%	62	1
3	0	24.2%	44	0
4	1	26.2%	54	0
4	0	18.1%	43	0
5	1	26.2%	54	0
5	0	17.9%	43	0
6	1	25.9%	51	0
6	0	17.6%	41	0

This Multi-component signature system generate signature shown in Table6. The limitation of Hancock is its low coverage. One potential explanation for this result is that malware distribution strategy has been evolved by malware authors recently. However, this stimulate the Apposcopy's signature generating method of modifying source code to tag potential hazzardous functions rather than just use probability models.

3.3 On Lightweight Mobile Phone Application Certification

3.3.1 *Introduction*

The former two papers focus on PC platform. But on Android, malware also have to be detected. Different from previous platforms, Android apps depend on markets to distribute. Thus, Kirin certification system is built to detect android malwares with various security rules. Besides, Kirin tool (1) provides a methodology for retrofitting security requirements in Android; (2) provides a practical method of performing lightweight certification of applications at install time.

3.3.2 *Structure of Kirin*

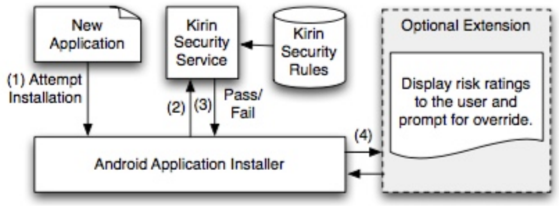


Figure 1: Kirin based software installer

Figure1 shows the Android's existing Security framework with Kirin work on it during app installation. The Kirin system will collect the security rules and the behaviors the app might have against the rules. With this system, the use can make more informed decision. Kirin relies on well constructed security rules which is one of the main concern of this paper. Another contribution is the security language combine with Kirin called Kirin security language(KSL).

3.3.3 *Kirin Security Rules*

Kirin identify security requirements following the steps in Figure3. (1)Instead of identifying assets from functional requirements, the author extract them from the features on the Android platform; (2) Kirin carefully study each asset to specify corresponding functional descriptions; (3) For each asset, Kirin must determine which goals are or not appropriate; (4) The system define security requirements from the threat descriptions; (5) Reach the goal to identify potentially dangerous configurations at install time.

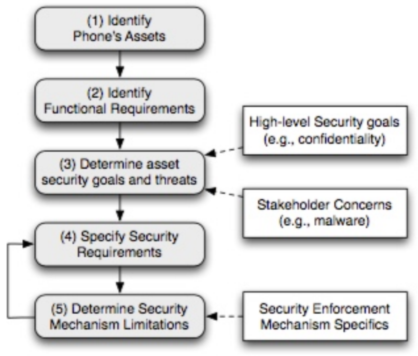
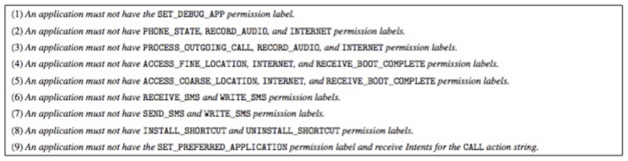

(1) Identify Phone's Assets
(2) Identify Functional Requirements
High-level Security goals (e.g., confidentiality)
(3) Determine asset security goals and threats
Stakeholder Concerns (e.g., malware)
(4) Specify Security Requirements
(5) Determine Security Mechanism Limitations
Security Enforcement Mechanism Specifics

Figure 3: Procedure for requirements identification

The security Rules on single permission and multiple permissions help Kirin to generate regularions to mitigate malware. Figure4 gives the details of the rules.



(1) *An application must not have the* SET_DEBUG_APP *permission label.*
(2) *An application must not have* PHONE_STATE, RECORD_AUDIO, *and* INTERNET *permission labels.*
(3) *An application must not have* PROCESS_OUTGOING_CALL, RECORD_AUDIO, *and* INTERNET *permission labels.*
(4) *An application must not have* ACCESS_FINE_LOCATION, INTERNET, *and* RECEIVE_BOOT_COMPLETE *permission labels.*
(5) *An application must not have* ACCESS_COARSE_LOCATION, INTERNET, *and* RECEIVE_BOOT_COMPLETE *permission labels.*
(6) *An application must not have* RECEIVE_SMS *and* WRITE_SMS *permission labels.*
(7) *An application must not have* SEND_SMS *and* WRITE_SMS *permission labels.*
(8) *An application must not have* INSTALL_SHORTCUT *and* UNINSTALL_SHORTCUT *permission labels.*
(9) *An application must not have the* SET_PREFERRED_APPLICATION *permission label and receive Intents for the* CALL *action string.*

Figure 4: Sample Kirin security rules to mitigate malware

3.3.4 *Kirin Security Language(KSL)*

KSL helps to generate rules with simple syntax in logic. The whole language defines rule-set, restrict and constant of applications like Figure5.

The KSL semantics is focusing on analysing manifest file correspond to a set of KSL rules. If the application pass Kirin test, The app will install or the installation should be blocked.

Compare to Apposcopy, KSL is much different with SPEC Malware Language, but this might stimulate the invention of SPEC. They are much different of rules because SPEC generate rules from semantics and KSL is just from permission use.

$$\begin{array}{rcll}
\langle\text{rule-set}\rangle & ::= & \langle\text{rule}\rangle \mid \langle\text{rule}\rangle\ \langle\text{rule-set}\rangle & (1)\\
\langle\text{rule}\rangle & ::= & \text{“}\texttt{restrict}\text{”}\ \langle\text{restrict-list}\rangle & (2)\\
\langle\text{restrict-list}\rangle & ::= & \langle\text{restrict}\rangle \mid \langle\text{restrict}\rangle\ \text{“}\texttt{and}\text{”}\ \langle\text{restrict-list}\rangle & (3)\\
\langle\text{restrict}\rangle & ::= & \text{“}\texttt{permission [}\text{”}\ \langle\text{const-list}\rangle\ \text{“}\texttt{]}\text{”} \mid & \\
 & & \text{“}\texttt{receive [}\text{”}\ \langle\text{const-list}\rangle\ \text{“}\texttt{]}\text{”} & (4)\\
\langle\text{const-list}\rangle & ::= & \langle\text{const}\rangle \mid \langle\text{const}\rangle\ \text{“}\texttt{,}\text{”}\ \langle\text{const-list}\rangle & (5)\\
\langle\text{const}\rangle & ::= & \text{“}\texttt{'}\text{”}\texttt{[A-Za-z0-9\_.]+}\text{“}\texttt{'}\text{”} & (6)
\end{array}$$

Figure 5: KSL syntax in BNF.

3.3.5 *Kirin as a service*

Kirin application can be implemented as a service in Android app background. While installing applications, the installer passes the file path to the apk file to the RPC interface. Then, Kirin parses the package to extract the security configuration stored in the package manifest. The PackageManager APIs provide the necessary information. Different from Apposcopy, Kirin works on Android mobiles effeciently while Apposcopy work on a server to scan all applications.

3.3.6 Evaluation on Kirin

Practical security rules might let mitigate malware to be installed.Kirin’s certification technique conservatively detects dangerous functionality, and may reject legitimate applications. Kirin’s conservative certification technique only requires user involvement for approximately 1.6% of applications. Thus, Kirin seems to be very effective at mitigating malware.

3.4 TaintDroid: An Information-Flow Tracking System for Realtime Privacy

3.4.1 Introduction

TaintDroid is dynamic taint analyses that track information flow by instrumenting the Dalvik VM.

Is an extension to the Android mobile-phone platform that tracks the flow of privacy sensitive data through third-party applications. TaintDroid assumes that downloaded, third-party applications are not trusted, and monitors–in realtime–how these applications access and manipulate users’ personal data.

3.4.2 Technique overview

They leveraged Android’s virtualized architecture to integrate fTheir granularities of taint propagation: variable-level, method-level, message-level, and file-level. Though the individual techniques are not new, their contributions lie in the integration of these techniques and in identifying an appropriate trade-off betTheyen performance and accuracy for resTheirce constrained smartphones.

They use dynamic taint analysis (also called “taint tracking”) to monitor privacy sensitive information on smartphones. Sensitive information is first identified at a taint sTheirce, where a taint marking indicating the information type is assigned. Dynamic taint analysis tracks how labeled data impacts other data in a way that might leak the original sensitive information. This tracking is often performed at the instruction level. Finally, the impacted data is identified before it leaves the system at a taint sink (usually the network interface).

3.4.3 Comparing to previous work

Existing taint tracking approaches have several limitations. First and foremost, approaches that rely on instruction-level dynamic taint analysis using whole system emulation incur high performance penalties. Instruction-level instrumentation incurs 2-20 times slowdown in addition to the slowdown introduced by emulation, which is not suitable for real-time analysis. Second, developing accurate taint propagation logic has proven challenging for the x86 instruction set. Implementations of instruction-level tracking can experience taint explosion if the stack pointer becomes falsely tainted and taint loss if complicated instructions such as CMPXCHG, REP MOV are not instrumented properly. While most smartphones use the ARM instruction set, similar false positives and false negatives could arise.

3.4.4 Result

Experiments with TaintDorid for Android show that tracking incurs a runtime overhead of less than 14% for a CPU-bound microbenchmark. Moreover, interactive third-party applications can be monitored with negligible perceived latency.

They evaluated the accuracy of TaintDroid using 30 randomly selected, popular Android applications that use location, camera, or microphone data. TaintDroid correctly flagged 105 instances in which these applications transmitted tainted data; of the 105, They determined that 37 Theyre clearly legitimate. TaintDroid also revealed that 15 of the 30 applications reported users’ locations to remote advertising servers. Seven applications collected the device ID and, in some cases, the phone number and the SIM card serial number. In all, two-thirds of the applications in Their study used sensitive data suspiciously. Their findings demonstrate that TaintDroid can help expose potential misbehavior by third-party applications.

3.5 Dissecting Android Malware: Characterization and Evolution

3.5.1 Introduction

In this paper, they focus on the Android platform and aim to systematize or characterize existing Android malware. Particularly, with more than one-year effort, they have managed to collect more than 1,200 malware samples that cover the majority of existing Android malware families, ranging from their debut in August 2010 to recent ones in October 2011. In addition, they systematically characterize them from various aspects, including their installation methods, activation mechanisms as well as the nature of carried malicious payloads.

we present a systematic characterization of existing Android malware, ranging from their installation, activation, to the carried malicious payloads.

3.5.2 Result

3.5.2.1 Malware Installation

1)Repackaging

Repackaging is one of the most common techniques malware authors use to piggyback malicious payloads into popular applications (or simply apps). In essence, malware authors may locate and download popular apps, disassemble them, enclose malicious payloads, and then re-assemble and submit the new apps to official and/or alternative Android Markets. Users could be vulnerable by

being enticed to download and install these infected apps.

2)Update Attack

The first technique typically piggybacks the entire malicious payloads into host apps, which could potentially expose their presence. The second technique makes it difficult for detection. Specifically, it may still repackage popular apps. But instead of enclosing the payload, it only includes an update component that will fetch or download the malicious payloads at runtime. As a result, a static scanning of host apps may fail to capture the malicious payloads. In our dataset, there are four malware families, i.e., BaseBridge, DroidKungFuUpdate, AnserverBot, and Plankton, that adopt this attack.

3)Drive-by Download

The third technique applies the traditional drive-by download attacks to mobile space. Though they are not directly exploiting mobile browser vulnerabilities, they are essentially enticing users to download “interesting” or “feature-rich” apps. In our collection, we have four such malware families, i.e., GGTracker, Jifake, Spitmo and ZitMo. The last two are designed to steal user’s sensitive banking information.

3.5.2.2 Activation

Next, they examine the system-wide Android events of interest to existing Android malware. By registering for the related system-wide events, an Android malware can rely on the built-in support of automated event notification and callbacks on Android to flexibly trigger or launch its payloads. For simplicity, we abbreviate some frequentlyused Android events in Table 5.

Abbreviation	Events	Abbreviation	Events	Abbreviation	Events
BOOT (Boot Completed)	BOOT_COMPLETED	SMS (SMS/MMS)	SMS_RECEIVED WAP_PUSH_RECEIVED	NET (Network)	CONNECTIVITY_CHANGE PICK_WIFI_WORK
CALL (Phone Events)	PHONE_STATE NEW_OUTGOING_CALL	USB (USB Storage)	UMS_CONNECTED UMS_DISCONNECTED	MAIN (Main Activity)	ACTION_MAIN
PKG (Package)	PACKAGE_ADDED PACKAGE_REMOVED PACKAGE_CHANGED PACKAGE_REPLACED PACKAGE_RESTARTED PACKAGE_INSTALL	BATT (Power/Battery)	ACTION_POWER_CONNECTED ACTION_POWER_DISCONNECTED BATTERY_LOW BATTERY_OKAY BATTERY_CHANGED_ACTION	SYS (System Events)	USER_PRESENT INPUT_METHOD_CHANGED SIG_STR SIM_FULL

Table 5. THE (ABBREVIATED) ANDROID EVENTS/ACTIONS OF INTEREST TO EXISTING MALWARE

3.5.2.3 Malicious Payloads

1) Privilege Escalation The Android platform is a complicated system that consists of not only the Linux kernel, but also the entire Android framework with more than 90 open-source libraries included, such as WebKit, SQLite, and OpenSSL. The complexity naturally introduces software vulnerabilities that can be potentially exploited for privilege escalation. In Table IV, we show the list of known Android platform-level vulnerabilities that can be exploited for privilege exploitations. Inside the table, we also show the list of Android malware that actively exploit these vulnerabilities to facilitate the execution of their payloads.

2) Remote Control During our analysis to examine the remote-control functionality among the malware payloads, we are surprised to note that 1,172 samples (93.0%) turn the infected phones into bots for remote control. Specifically, there are 1,171 samples that use the HTTP-based web traffic to receive bot commands from their C&C servers.

3) Financial Charge Beside privilege escalation and remote control, we also look into the motivations behind malware infection. In particular, we study whether malware will intentionally cause financial charges to infected users.

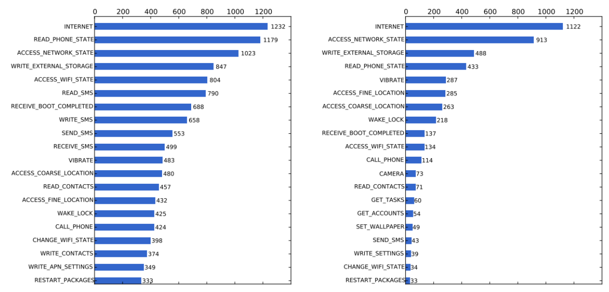
AN OVERVIEW OF EXISTING ANDROID MALWARE (PART II: MALICIOUS PAYLOADS)

	Privilege Escalation					Remote Control		Financial Charges			Personal Information Stealing		
	Exploid	RATC/ Zimperlich	Ginger Break	Asroot	Encrypted	NET	SMS	Phone Call	SMS	Block SMS	SMS	Phone Number	User Account
ADRD						✓							
AnserverBot						✓			✓†				
Asroot				✓									
BaseBridge		✓				✓		✓	✓†	✓			
BeanBot						✓		✓	✓†	✓		✓	
BgServ						✓			✓†	✓		✓	
CoinPirate						✓			✓†	✓	✓		
Crusewin						✓			✓	✓	✓		
DogWars									✓				
DroidCoupon		✓				✓							
DroidDeluxe		✓											
DroidDream	✓	✓				✓							
DroidDreamLight						✓							✓
DroidKungFu1	✓	✓			✓	✓						✓	
DroidKungFu2	✓	✓			✓	✓						✓	
DroidKungFu3	✓	✓			✓	✓						✓	
DroidKungFu4						✓							
DroidKungFu5	✓	✓			✓	✓						✓	
DroidKungFuUpdate													
Endofday						✓			✓			✓	
FakeNetflix													✓
FakePlayer									✓‡				
GamblerSMS											✓		
Geinimi						✓		✓	✓†	✓	✓	✓	
GGTracker									✓‡	✓	✓	✓	
GingerMaster			✓			✓						✓	
GoldDream						✓		✓	✓†		✓	✓	
Gone60											✓		
GPSSMSSpy									✓				
HippoSMS									✓‡	✓			
Jifake									✓‡				
jSMSHider						✓			✓†	✓		✓	
KMin						✓			✓†	✓			
Lovetrap									✓†	✓			
NickyBot							✓		✓		✓		
Nickyspy						✓			✓		✓		
Pjapps						✓			✓†	✓		✓	
Plankton						✓							
RogueLemon						✓			✓†	✓	✓		
RogueSPPush									✓‡	✓			
SMSReplicator									✓		✓		
SndApps													✓
Spitmo						✓			✓†	✓	✓	✓	
TapSnake													
Walkinwat									✓				
YZHC						✓			✓‡	✓		✓	
zHash	✓												
Zitmo											✓		
Zsone									✓‡	✓			
number of families	*6*	*8*	*1*	*1*	*4*	*27*	*1*	*4*	*28*	*17*	*13*	*15*	*3*
number of samples	*389*	*440*	*4*	*8*	*363*	*1171*	*1*	*246*	*571*	*315*	*138*	*563*	*43*

Table 6. AN OVERVIEW OF EXISTING ANDROID MALWARE (PART II: MALICIOUS PAYLOADS).

3.5.2.4 Permission Uses

For Android apps without root exploits, their capabilities are strictly constrained by the permissions users grant to them. Therefore, it will be interesting to compare top permissions requested by these malicious apps in the dataset with top permissions requested by benign ones. To this end, we have randomly chosen 1260 top free apps downloaded from the official Android Market in the first week of October, 2011. The results are shown in Figure below.



(a) Top 20 Permissions Requested By 1260 Malware Samples

(b) Top 20 Permissions Requested by 1260 Top Free (Benign) Apps on the Offical Android Market

Figure 5. AN OVERVIEW OF EXISTING ANDROID MALWARE (PART II: MALICIOUS PAYLOADS).

3.6 Apposcopy: Semantics-Based Detection of Android

3.6.1 Introduction

Apposcopy is a new semantics-based approach for identifying a prevalent class of Android malware that steals private user information. Apposcopy incorporates (i) a highlevel language for specifying signatures that describe semantic characteristics of malware families and (ii) a static analysis for deciding if a given application matches a malware signature. The signature matching algorithm of Apposcopy uses a combination of static taint analysis and a new form of program representation called Inter-Component Call Graph to efficiently detect Android applications that have certain control- and data-flow properties.

3.6.2 Technique overview

The signature-based specification language provided by Apposcopy allows specifying two types of semantic properties |control-flow and data-flow|of Android applications. An example of a control-flow property is that the malware contains a broadcast receiver which launches a service upon the completion of some system event. An example of a data flow property is that the malware reads some private dataof the device and sends it through a designated sink (e.g., Internet, SMS message, etc.). To match the signatures specified in this language, Apposcopy's static analysis relies on two key ingredients. First, they construct a new high-level representation of Android applications called the inter-component callgraph (ICCG), which is used to decide whether an Android application matches the control flow properties specified in the signature. Second, Apposcopy incorporates a static taint analysis which is used for deciding whether a given application matches a specified data-flow property.

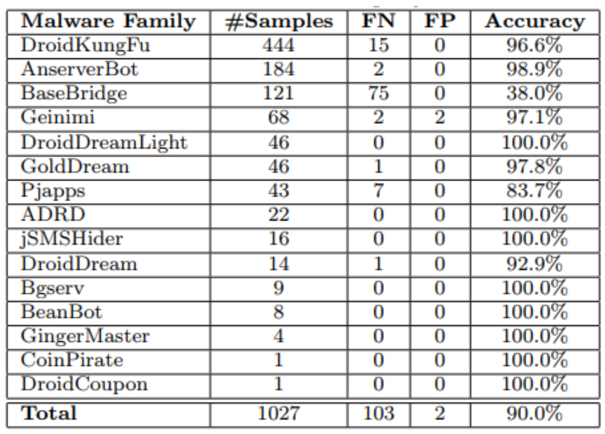
3.6.3 Comparing to previous work

Taint analyses are capable of exposing applications that leak private user information. unfortunately, since many benign apps also need access to sensitive data to perform their advertised functionality, not every app that leaks user information can be classified as malware. For instance, an email client application will necessarily "leak" email addresses of the user's contacts to perform its functionality. Thus, taint analyses cannot automatically distinguish benign apps from malware, and a security auditor must invest significant effort to determine whether a given information flow constitutes malicious behavior.

Signature-based malware detectors, including commercial virus scanners, classify a program as malware if it contains a sequence of instructions that is matched by a regular expression. As shown in a recent study, malware detectors that are based on syntactic low-level signatures can be easily circumvented using simple program obfuscations. Hence, these malware signatures must be frequently updated as new variants of the same malware family emerge.

3.6.4 Reslut

In Their first experiment, they evaluate the effectiveness of Apposcopy on 1027 malware instances from the Android Malware Genome project, which contains real malware collected from various sTheirces, including Chinese and Russian third-party app markets. All of these malicious applications belong to known malware families, such as DroidKungFu, Geinimi, and GoldDream. To perform this experiment, they manually wrote specifications for the malware families included in the Android Malware Genome Project. For this purpose, they first read the relevant reports where available and inspected a small number (1-5) of instances for each malware family. Table 7 presents the results of this experiment. The first column indicates the malware family, and the second column shows the number of analyzed instances of that malware family. The next two columns show the number of false negatives (FN) and false positives (FP) respectively.



Malware Family	#Samples	FN	FP	Accuracy
DroidKungFu	444	15	0	96.6%
AnserverBot	184	2	0	98.9%
BaseBridge	121	75	0	38.0%
Geinimi	68	2	2	97.1%
DroidDreamLight	46	0	0	100.0%
GoldDream	46	1	0	97.8%
Pjapps	43	7	0	83.7%
ADRD	22	0	0	100.0%
jSMSHider	16	0	0	100.0%
DroidDream	14	1	0	92.9%
Bgserv	9	0	0	100.0%
BeanBot	8	0	0	100.0%
GingerMaster	4	0	0	100.0%
CoinPirate	1	0	0	100.0%
DroidCoupon	1	0	0	100.0%
Total	1027	103	2	90.0%

Table 7. Evaluation of Apposcopy on malware from the Android Malware Genome project.

In a second experiment, they evaluate Apposcopy on thousands of apps from Google Play. Since these applications are available through the official Android market rather than less reliable third-party app markets, they would expect a large majority of these applications to be benign. Hence, by running Apposcopy on Google Play apps, they can assess whether Their high-level signatures adequately differentiate benign applications from real malware. In their experiment, among the 11,215 apps analyzed by Apposcopy, only 16 of them theyre reported as malware. And 13 of them are indeed malware.

To substantiate their claim that Apposcopy is resilient to code transformations, they compare the detection rate of Apposcopy with other anti-virus tools on obfuscated versions of known malware. Compare to other anti-virus tools, Apposcopy achieved 100% accuracy.

In addition to comparing Apposcopy with commercial antivirus tools, they also compared Apposcopy against Kirin, which is the only publicly available research tool for Android malware detection. Result shows that Apposcopy outperformed Kirin on both FP and FN by far.

5. RESULT

From all six papers we learn that of semantics-based approach is the prevailing method for android malware detection because syntax-based approach can be circumvented by obfuscation. We also learn that taint analysis (both dynamic and static based) can give us the data flow information and ICCG can tell us control flow information,

We now also have a general understanding of android malware. We know their behavior pattern and how they perform malicious actions,

6. CONCLUTION

From six papers we surveyed, we came to a conclusion that semantic-based static analysis is the prevailing approach to detect android malware. We learn that control flow and data flow information are two patterns that can be used as patterns of a given application. By analysis a set of known malwares' behavior patterns, we can generalize signatures of those malwares and use that signature to detected potential malwares.

7. REFERENCES

[1] Zhou, Yajin, and Xuxian Jiang. "Dissecting android malware: Characterization and evolution." S&P 2012.

[2] The Activity Lifecycle | Android Developershttps://developer.android.com/guide/components/activities/activity-lifecycle.html

[3] Enck, William, Machigar Ongtang, and Patrick McDaniel. "On lightweight mobile phone application certification." CCS 2009.

[4] W. Enck, P. Gilbert, B. gon Chun, L. P. Cox, J. Jung, P. McDaniel, and A. Sheth. TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones. OSDI 2010.

[5] M. Christodorescu, S. Jha, S. A. Seshia, D. X. Song, and R. E. Bryant. Semantics-aware malware detection. S&P 2005.

[6] K. Griffin, S. Schneider, X. Hu, and T. cker Chiueh. Automatic generation of string signatures for malware detection. RAID 2009.

[7] Yu Feng , Saswat Anand , Isil Dillig , Alex Aiken, Apposcopy: semantics-based detection of Android malware through static analysis, Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, November 16-21, 2014.