DroidTrace: A Ptrace Based Android Dynamic Analysis System with Forward Execution Capability

Min Zheng, Mingshen Sun, John C.S. Lui
The Chinese University of Hong Kong
{mzheng,mssun,cslui}@cse.cuhk.edu.hk

***Abstract*—Android, being an open source smartphone operating system, enjoys a large community of developers who create new mobile services and applications. However, it also attracts malware writers to exploit Android devices in order to distribute malicious apps in the wild. In fact, Android malware are becoming more sophisticated and they use advanced "*dynamic loading*" techniques like Java reflection or native code execution to bypass security detection. To detect dynamic loading, one has to use dynamic analysis. Currently, there are only a handful of Android dynamic analysis tools available, and they all have shortcomings in detecting dynamic loading. The aim of this paper is to design and implement a dynamic analysis system which allows analysts to perform systematic analysis of dynamic payloads with malicious behaviors. We propose "*DroidTrace*", a ptrace based dynamic analysis system with forward execution capability. Our system uses ptrace to monitor selected system calls of the target process which is running the dynamic payloads, and classifies the payloads behaviors through the system call sequence, e.g., behaviors such as file access, network connection, inter-process communication and even privilege escalation. Also, DroidTrace performs "physical modification" to trigger different dynamic loading behaviors within an app. Using DroidTrace, we carry out a large scale analysis on 36,170 dynamic payloads in 50,000 apps and 294 malware in 10 families (four of them are zero-day) with various dynamic loading behaviors.**

I. Introduction

In the past few years, Android malware are becoming more sophisticated. They use advanced *dynamic loading* (or dynamic library loading) techniques like Java reflection and native code execution to bypass security detection. Note that dynamic library loading is a common technique used by legitimate Android apps developers. The main reason is that Java code can be easily disassembled, in order to protect the software, Android developers use Java reflection to dynamically load encrypted `jar` packages, or they can use the Native Development Kit (NDK) to develop applications in C or C++. Dynamic library loading makes it more difficult for others to reverse engineer the code, hence protecting the intellectual property of the developers, but it also adds more complexity on security analysis since malware writers also deploy this technique in their malware development.

Briefly speaking, dynamic loading is a mechanism in which a program can load a library (or dynamic payloads) into memory at runtime, retrieve the addresses of functions contained in the dynamic library, execute these functions and unload the library from the memory upon completion. It is a popular and effective way to protect Android applications (or malware). To illustrate how common this technique is being used, we carried out the following study: As of March 2013, we have collected 50,000 latest apps from the Google Play and third-party markets. We have found that 32.8% (16,396 apks) of apps have dynamic loading behaviors: 21.7% (10,841 apks) of apps have dynamic library which is written in C/C++ via NDK,and 15.7% (7,836 apks) of apps contain dynamic payloads like `elf`, `so`, `jar`, `apk` or `dex` files. Table I summarizes our study. Note that because dynamic payloads can be downloaded from the Internet when the applications execute, the actual number of dynamic payloads is far more than what we are reporting here. Yet, the take home message is that dynamic library loading is a common and popular technique for app development. Note that there can be more than one dynamic payload file in an application file, and this conservative statistical estimates already reveal the popularity of using dynamic loading.

	Applications (50,000 Apps)	
	# of APK	**# of Dynamic Payload**
`so` file	10,841	24,596
`elf` file	472	841
`dex` (`jar, apk`) file	7,364	10,733
Total Number	18,677	36,170
Unique MD5 Number	16,396	12,712

TABLE I. Statistics of Dynamic Payloads in 50,000 Apps

To detect and examine malicious dynamic loading, one needs to perform static and dynamic analysis of an apps. Both of these techniques have advantages and disadvantages. Static analysis is usually faster and can give analysts a more comprehensive code coverage in analyzing the app since it can explore different execution paths. However, static analysis is not effective on dynamic loading. For example, most of the recent Android analytic systems [1], [4], [10], [22] stated their inefficiency in against dynamic loading. Dynamic analysis, on the other hand, is useful to study the *runtime behavior* of an app. But one drawback is in determining the code path in order to *trigger* the dynamic loading. Furthermore, because Android is a redesigned Linux system, Google adds many components, e.g., UI surface, binder communication, . . ., etc, into Android. Therefore, one cannot use the legacy Windows or Linux-based dynamic analysis systems. Currently, there are only a handful of Android dynamic analysis systems and they all have shortcomings in handling dynamic loading. For example, TaintDroid[3] is built on top of the Dalvik virtual machine, and it cannot handle dynamic payloads (e.g. `elf` or `so` file) since they run in native code level. DroidScope[20] is built on top of a hypervisor and it simulates the hardware environment so it cannot run on real smartphone device. Also, both of these systems cannot automatically execute the app under study, so analysts have to go through a tedious process of *manually triggering* different code paths to explore malicious behaviors.

In this paper, we propose "*DroidTrace*", a ptrace based dynamic analysis system with forward execution capability, and we focus on how to use DroidTrace to explore the behavior of dynamic payloads. Ptrace (Process Trace) is a system call and we use it to observe and control the execution of another process. In DroidTrace, we use ptrace to monitor system calls of the target process which is running the dynamic payloads. Since the ptrace system call is part of the linux kernel, so DroidTrace can monitor all the behaviors of dynamic payloads, both at the java code level or at the native code level (i.e., C or C++). In addition, DroidTrace can be executed on real devices so analysts can test different hardware platforms without resorting to emulation. Last but not least, DroidTrace can perform *physical modification* to trigger different dynamic loading behaviors.

Contributions: The contributions of our work are:

- We present the design and implementation of a ptrace-based dynamic analysis system which can monitor all the behaviors of dynamic payloads on both java and native code level. In addition, DroidTrace can be executed on real devices for all Android versions.
- We propose a forward execution methodology called "physical modification" on DroidTrace so one can automatically trigger most of the dynamic loading behaviors and manually trigger all dynamic loading behaviors. This significantly enhances the security analysis process to discover malicious behaviors.
- To show the effectiveness of DroidTrace, we carried out large scale experiments on 36,170 dynamic payloads in 50,000 apps and 294 malware in 10 families (four of them are zero-day) which have dynamic loading behavior.

II. Analysis Methodology

Let us present our analysis methodology. The main idea is as follows: DroidTrace first uses static analysis to discover functions which have the dynamic loading behavior, i.e., running an `elf` or `so` file, or downloading an executable from the Internet. Then it sets these functions as "*targets*" and generates the *application control flow graph* (ACFG) so that the system can explore the possible execution paths to these functions. By using the ACFG, DroidTrace generates the forward execution paths for each target. Because some information or malicious behaviors can only be obtained at runtime, DroidTrace will automatically run the application via each forward execution path and then use the ptrace based dynamic analysis tool to analyze the dynamic payloads (e.g., `so` file, `elf` file, and `dex` file). In general, the methodology consists of four steps:

A. Step 1: Set Functions as the Targets

In this step, DroidTrace disassembles the `dex` file of the application into smali code. The smali syntax provides information of how variables and methods are being invoked. Then DroidTrace determines the functions which have the dynamic loading behavior (i.e., downloading a file from the Internet or calling an `elf` or `so` file) as the targets and the system will search for these functions in the smali code level. Note that because some dynamic payloads may be invoked by other dynamic payloads, so DroidTrace also handles the *inter-calling* between dynamic payloads. The invoking relationships between various components is illustrated in Figure 1. Let us explain how we use the inter-calling relationships to explore all dynamic behaviors within an app.

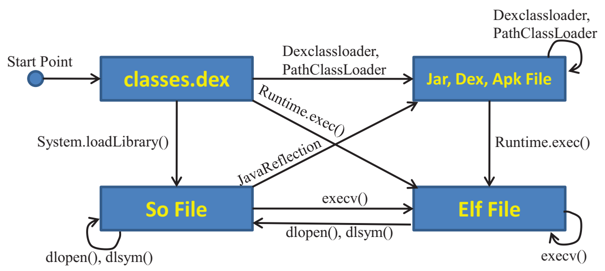

Fig. 1. Relationships between Dynamic Payloads Components

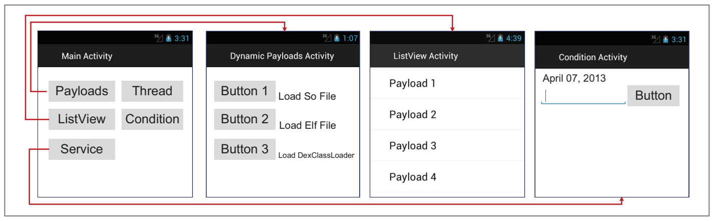

Fig. 2. Screen Capture of an App under DroidTrace

To illustrate our methodology, consider an example app which is depicted in Figure 2. The application starts with the `Main Activity` which has five buttons. By clicking the `Payloads`, `ListView` or `Condition` button, the application will jump to the corresponding activity. By clicking the `Thread` or `Service` button, the application will start a thread or a service running in the background and trigger the dynamic loading behavior. The application also has a broadcast receiver which receives the `android.intent.action.BOOT_COMPLETED` message. After receiving the message, the receivier will start a service which has the dynamic loading behavior. The dynamic payloads activity in our example application has three buttons: `elf file`, `so file`, and `DexClassLoader`. When the user clicks the `elf file` button, the application will run an `elf` executable with a file operation behavior. For the `so File` button, the application will load a `so` file which has a networking behavior. For the `DexClassLoader` button, the application will dynamically load a `dex` file which will in turn run an `elf` file with a privilege elevation behavior.

B. Step 2: Generate Application Control Flow Graph

After setting the targeted functions, DroidTrace generates a *control flow graph* (CFG) for each function and a *function call graph* (FCG) for the application. Then DroidTrace combines them together into an *application control flow graph* (ACFG). By using the ACFG, DroidTrace can determine various execution paths from the start point to different targets. Here, we need to address several challenges. Firstly, unlike the `main()` function in C language, there are many entries in an Android app: entries via `activity`, `service`, `broadcast receiver`, ..., etc. Hence, the ACFG will have several start entries. Secondly, DroidTrace not only needs to connect the function calls (e.g., method A in class B invokes method C in class D) together, but also needs to handle special components like threads and runnables, intents and receivers, buttons and button listeners, ..., etc. These special components do not have direct connection in the source code, but they can communicate through the Android API. Thirdly, some dynamic loading behaviors are triggered by specific conditions (e.g., date and event), hence, DroidTrace needs to consider the conditional executions in the ACFG.

1) Application Entries: In DroidTrace, the system parses the `AndroidManifest.xml` to get information of all entries (e.g., `activities`, `services`) of the `apk` file. Every Android application has an `AndroidManifest.xml` file in its root directory. It names the classes that implement each of the components and publishes their capabilities (e.g., which `Intent` messages they can handle).

2) Components Connections: In Android, components may not be directly connected but they can communicate through the Android API. In order to get the full *application control flow graph*, DroidTrace connects them based on the Android API mechanism. For example, user Input controls are the interactive components in an app's user interface. Android provides a variety of controls one can use in an application user-interface (UI), such as buttons, listview, seek bars, and so on. For example, when a user clicks a button, the `Button` object receives an on-click event. To handle this event, developers need to create a `View.OnClickListener` object and assign it to the button by calling the `setOnClickListener()`. In DroidTrace, the system connects all the UI components with their handlers in the ACFG. In this example, the system connects the click `button1` event with the related `OnClickListener`'s `onClick()` function.

3) Conditional Execution: In DroidTrace, the system handles the conditional execution like `if...else...` statement and `switch` statement. Firstly, DroidTrace records the conditional expressions and forks a new execution path for each condition. Secondly, DroidTrace determines which value in the conditional expressions is a constant value, and which is a variable value.

C. Step 3: Perform Forward Execution

Android application is UI-based, so analysts cannot simply provide input parameters to run the mobile app, but rather, they need to provide user events such as clicks, touches, or gestures to the related UI components on the smartphone screen. There are several ways to carry out the forward execution in Android. Google provides developers a *monkeyrunner* API[7] to simulate the user events. The monkeyrunner tool provides an API for writing Python programs that can control an Android device or emulator. However, monkeyrunner is complicated to integrate for auto-execution because the ID of the UI components can only be obtained at runtime. So it cannot determine the execution path before running. In addition, several researchers need to modify the Android emulator in order to dothe forward execution (e.g., [21]). In DroidTrace, we propose a *new* methodology which uses physical modification to perform the forward execution: For each forward execution path to a function with dynamic behavior, the system first disassembles the appliciaton, adds the forward trigger code and then generates a new and separated `apk` file. For each rebuilt `apk` file, DroidTrace installs the `apk` in the emulator, then the installed application will automatically run one of its dynamic loading functions and trigger the behavior of dynamic payloads. The advantage of using physical modification is the system only modified the `apk` file, so DroidTrace can run on both emulator and real android devices. In addition, the system can explore various execution paths before running. For perform such forward execution, we have to consider the following events:

Application Entries: Activity Manager (AM) is an Android SDK tool which can be used to start activities and services at the command line, or send intents to running applications. DroidTrace uses AM to start the target activities, services or broadcast receivers.

User Events: In DroidTrace, the system uses physical modification to achieve forward execution. The methodology is to modify the `apk` file before the application runs via techniques of reverse engineering. DroidTrace will add user events trigger codes into the `apk` file. For example, when a user clicks a button, the application will call the `onClick()` function of the corresponding `OnClickListener`. Therefore, DroidTrace adds the trigger code into the application by reverse engineering so that the application will automatically trigger the on-click button event without any human interactions, for example:

```
button1.setOnClickListener(clickListener);
//adding this line of code
clickListener.onClick(button1);
```

Conditional Execution: As we mentioned in Step 2 of generating the ACFG, DroidTrace will record the conditional expressions and fork a new execution path for each condition. In order to execute the conditional path, DroidTrace also uses the physical modification methodology to add the trigger code into the `apk` file.

D. Step 4: Suspicious Behavior Detection

Android is a derivative based on a modified Linux 2.6 with a Java programming interface. If any Android application needs to request services (e.g., accessing the flash drive), it has to rely on system calls provided by the kernel. Therefore, by monitoring the system calls of a process, DroidTrace can obtain the runtime information of the process. There are several ways to monitor system calls, for example, modifying the kernel by injecting the monitoring code[3], running the Android OS on a hypervisor to obtain the information of the underlying process[20], ..., etc. We choose to use `ptrace`, a system call that can control and observe the execution of another process. The advantage of using ptrace in dynamic analysis is that one does not need to modify the kernel, so future update by Google on Android OS will not affect DroidTrace. Furthermore, DroidTrace can be executed on real devices so analysts can precisely determine the impact of apps on a particular device.

The dynamic analysis module of DroidTrace is built on top of Strace[13]. Strace is a system call tracer for Linux which can print out the system calls made by another process/program. However, using Strace to do dynamic analysis has a number of problems. Firstly, a process usually makes many system calls. If we use Strace to monitor all system calls, the overhead will be unacceptable as the application will be running at an order of magnitude slower than the normal mode, hence, slowing down the analysis process. Secondly, most parameters of system calls are memory addresses. For example, the Binder, one of the most important inter-process communication (IPC) mechanisms in Android OS, uses `iotcl()` system call to transform the information using memory. Using Strace to do dynamic analysis cannot provide the structural information except the memory addresses, and it is futile to carry out dynamic analysis if the system does not parse the structure information at these addresses.

To overcome these problems, we propose a *behavior rule mechanism*. Based on this mechanism, DroidTrace classifies the system call sequence into different *behaviors*. This mechanism offers several advantages. Firstly, DroidTrace will not directly show all system calls information to the analysts because the raw output data are large and they may obscure the analysis. DroidTrace classifies the system calls information based on the behavior rule table, then outputs the suspicious behaviors of the dynamic payload. Secondly, for some special system calls (e.g., `ioctl()`), DroidTrace extracts the related information from the memory addresses. In addition, analysts can extend the behavior rules so to support more detailed analysis tasks, e.g., if a mobile app uses the `open()` system call to open the contact list file (`/data/data/com.android.providers.contacts-/databases/contacts2.db`), and then uses the `read()` system call to get the contact list. This can be classified as *accessing contact information behavior*. When this behavior is in the behavior rule table, DroidTrace will show this particular behavior of the dynamic payload (e.g., reading the contact information) to the analysts, so making the analysis more readable and insightful. Currently, DroidTrace supports four kinds of behaviors detection, they are: behavior on *(a) file operation*, (b) *network connection*, (c) *inter-process communication*, and (d) *privilege escalation*.

III. Case Study and Evaluation

In this section, we summarize the statistics of our large scale malware experiment and one case study to illustrate the effectiveness of DroidTrace.

A. *Statistics on large scale experiments*

To show the effectiveness of DroidTrace, we perform large scale experiments on 50,000 legitimate apps and 294 malware we collected which have dynamic loading behavior in 10 malware families (with four malware families being zero-day malware, meaning, they were just injected into the wild.) Table I shows the summary statistics of dynamic payloads of these 50,000 apps. Using DroidTrace, we first used static analysis to filter those applications without dynamic loading APIs and only kept 16,396 applications with dynamic loading behavior. Then the system used both static analysis and dynamic analysis techniques to trigger and monitor the behaviors of the dynamic payloads. Note that most of the processes are automatic, but some applications may require manual input (e.g., register and login for some mobile apps) in order to trigger the behaviors. Table II shows the detailed statistics of malware with dynamic loading behavior. For example, the ARAlarmReceiver, which is a zero-day malware family, has four samples and these samples have a payload which can install malware silently and steal user information with *root exploit*.

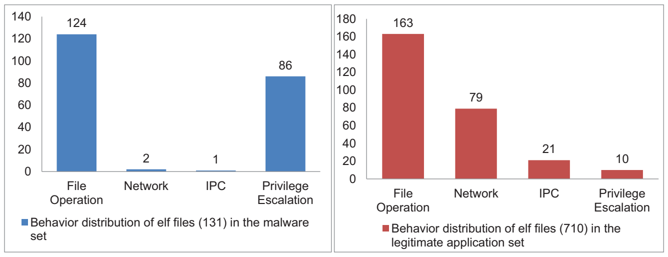

Fig. 3. Behavior Distribution of `elf` Files

There are 131 `elf` files in the malware set and 710 `elf` files in the legitimate application set. From Figure 3, we can see that both legitimate applications' and malware's `elf` files have various file operation behavior. However, most of malware's `elf` files have significantly more privilege escalation behavior but only a few `elf` files of legitimate applications (e.g., security software) have the privilege escalation behavior. Therefore, analysts should first focus on the `elf` files with privilege escalation behavior, because there is a high probability that it is a dynamic payload of the malware.

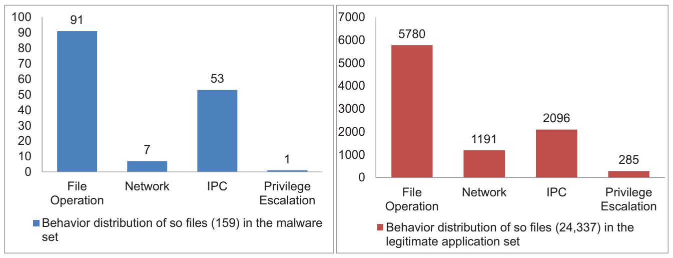

Fig. 4. Behavior Distribution of `so` Files

There are 159 `so` files in the malware set and 24,337 `so` files in the legitimate application set. From Figure 4, we can discover that the `so` files in the malware set and legitimate application set have a similar distribution (although the magnitude on each operation is different). The reason is that most malware are repackaged, and the malware writers

Families	Zero-day	File Number	Description of Malware Behavior
AndroidBot	No	1	The malware has *Root Exploit* and silently install a botnet app for SMS charging
ARAlarmReceiver	Yes	4	Botnet (for silent installation, steal user info, ..., etc.) with *Root Exploit*
BaseBridge	No	45	The malware has *Root Exploit* and silently install a botnet app for SMS charging
CIAppmaster	No	8	Botnet (for silent installation) with *Root Exploit*
CIAppmaster2 (variant)	Yes	1	Botnet (for silent installation) with *Root Exploit*
DroidKungFu	No	7	Botnet (for advertisement, silent installation, ..., etc.) with *Native Code* and *Root Exploit*
GASmart	No	3	Botnet (for silent installation) with *Root Exploit*
NetIusys828	Yes	1	Botnet (for silent installation) with *Native Code* and *Root Exploit*
CEPlugnew	Yes	1	Botnet (for SMS charging, silent installation, ..., etc.) with *DexClassloader*
Plangton	No	223	Botnet (for silent installation, steal user info, ..., etc.) with *DexClassloader*
All		294	

TABLE II. USING DROIDTRACE TO DISCOVER MALWARE WITH DYNAMIC LOADING BEHAVIOR

repackage the malicious java code into some legitimate applications which have `so` files. But this repackaging process has no effect on those `so` files. Therefore, most of the `so` files are benign. As a matter of fact, in our malware database, only the DoridKungfu family uses `so` file to defend against malware detection. This is the main reason why the distributions of `so` files in the malware set and legitimate application are very similar.

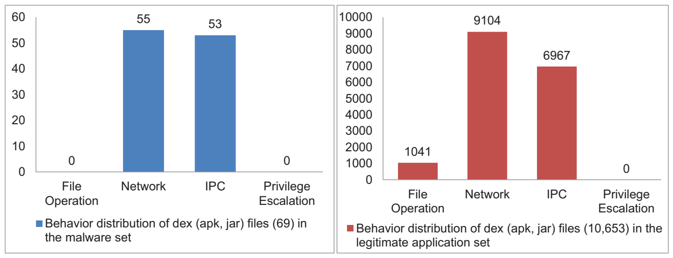

Fig. 5. Behavior Distribution of `dex`, `apk`, `jar` Files

There are 69 `dex` (or `apk,jar`) files in the malware set and 10,654 `dex` (or `apk,jar`) files in the legitimate application set. From Figure 5, we can discover that both malware's and legitimate application's `dex` (or `apk,jar`) files have high number of IPC and network behaviors but no privilege escalation behavior. For malware, malware writers tend to use Java reflection to dynamically load the malicious logic from the `dex` (or `apk,jar`) files in order to defend against malware detection. For legitimate applications, developers tend to use Java reflection to dynamically load the software plug-in components from the `dex` (or `apk,jar`) files for application extension. In addition, because of the Dalvik virtual machine (DVM) mechanism, the java code in the DVM does not have instructions to alter the `uid` of the DVM process, so there are no `dex` (or `apk,jar`) files with privilege escalation behavior.

B. Case Study: CEPlugnew

CEPlugnew is a zero-day malware which could not be detected by any anti-virus software until May, 2013. It is an advanced malware using `DexClassLoader` to dynamically decrypt and load the encrypted `jar` file. At first, we did not find any dynamic payloads in its attachment files. However, DroidTrace reported that this app has a dynamic loading behavior in its `com.a.a.a.d()` function. In order to trigger this function, DroidTrace explores the ACFG and uses the `AM` to send a `android.intent.action.USER_PRESENT` message to the malware. When the malware receives this message, it starts two services. One service dynamically registers a SMS receiver with the `2147483647` priority number (the largest priority number). Another service uses `DES` to decrypt the `rt2.jar` file in its `app_TYPE_JAR` folder. After the decryption, the malware generates a `appmgr.jar`, a real `jar` file in the `app_sim_index` folder. Then the malware uses `DexClassLoader` to dynamically load the `appmgr.jar` and reflects several methods in this `appmgr.jar`.

We then use the ptrace in DroidTrace to attach to the process of the dynamic payload and then find that `appmgr.jar` has several behaviors. First, the malware decrypts a configure file, `smsrpt.i`, from the attached `config` file and stores it in its `service` folder. Second, based on the `smsrpt.i` configure file, the malware uses the `imsi` (International Mobile Subscriber Identity) as the id to request a command from several remote servers (e.g. 42.121.120.*, 42.121.98.*). The most important command is to send some codes to some value-added telecom service numbers which belong to the hackers. For example, a user sends a SMS message "001" to the number "10086", then the user will receive newspapers everyday and pay this service for one dollar per month. Thirdly, the malware checks whether the smartphone is rooted or not. If the smartphone is rooted, the malware remounts the `/system` folder, and replaces the `xsu` with its own `xsu` file in its attachment folder. After that, the malware can silently download and install any application without any notification to the user. The detail of ptrace based dynamic analysis results can be referred to Table III.

IV. Related Work

Two common methodologies to detect and analyze malware are static [2], [18] and dynamic analysis [14], [16].

Static Analysis: When the Android malware started propagating in 2011, malware researchers mainly focused on permission analysis. Stowaway[5] provides a detailed table of permissions for each API call. Felt et al. [6] shows the attacks and defenses on the permission system on Android. After the prevailing of Android malware, researchers disassemble the applications and reveal the malicious codes by reverse engineering tools like smali and ded [12], [4]. Based on disassembled codes of malware, DroidMOSS [22] can detect repackaged applications by using fuzzy hashing. With these

System Authorized File Access - /data/data/com.example.plugnew/service/smsrpt.i
mkdir "/data/data/com.example.plugnew/service", 0700
open "/data/data/com.example.plugnew/service/smsrpt.i", O_CRE | O_EXC
write "configurl=http://42.121.120.87:9800/reg/reg.jsp?...."

System Unauthorized File Access - /system/bin/xsu
execve "/system/bin/mount", ["mount", "-o", "remount,rw", "-t", "yaffs2", "/dev/block/mtdblock3", "/system"]
execve "/system/bin/cp", ["cp", "/data/data/com.example.plugnew/service/xsu", "/system/bin/xsu"]

Network Connection - Http to 42.121.120.87
getsockname port=htons(59509), addr=inet_addr("0.0.0.0")
connect port=htons(9800), addr=inet_addr("42.121.120.87")
write info="GET /reg/reg.jsp?version=40005..."

IPC - Getting IMSI
ioctl com.android.internal.telephony.IPhoneSubInfo get IMSI

Privilege Escalation
fork child pid = 351
execve "/system/xbin/su"
getuid32 = 0

TABLE III. RESULTS OF DROIDTRACE ANALYSIS ON CEPLUGNEW

static information, Grace et al. developed RiskRanker [9] to rank the suspicious applications and attempted to figure out the zero-day malware. And there are other static analysis systems [8], [23], [24], [10] to detect the vulnerabilities on Android system. In general, all these static analysis systems cannot handle the dynamic payloads and obtain the runtime information.

Dynamic Analysis: In Section I, we discussed some dynamic analysis systems [3], [20], [19] for Android system. Both DroidScope [20] and CopperDroid [15] are based on hypervisor to monitor system behaviors. TaintDroid [3] is a realtime analysis system to monitor the privacy leakage. And some systems (e.g. [17], [11]) utilize TaintDroid to enhanced dynamic analysis. But TaintDroid can not monitor the malicious behaviors implemented by native code, which is part of the dynamic behavior we aim to address.

V. Conclusion

To study the malicious behaviors of malware with dynamic payloads, we propose DroidTrace, a ptrace based Android dynamic analysis system with forward execution. The system uses ptrace to monitor the system calls of the target process which is running the dynamic payloads and classifies the payloads behaviors through the system call sequence. In addition, DroidTrace performs forward execution so to trigger different dynamic loading behaviors. We demonstrate the effectiveness of DroidTrace by carrying out large scale experiments on 50,000 legitimate apps and 294 malware in 10 families (four of them are zero-day) which have dynamic loading behavior.

REFERENCES

[1] K. Z. Chen, N. Johnson, V. D'Silva, S. Dai, K. MacNamara, T. Magrino, E. X. Wu, M. Rinard, and D. Song. Contextual policy enforcement in android applications with permission event graphs. In *Proceedings of NDSS 2013*, February 2013.

[2] M. Christodorescu and S. Jha. Static analysis of executables to detect malicious patterns. In *Proceedings of SSYM'03*, 2003.

[3] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, OSDI'10, 2010.

[4] W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri. A study of android application security. In *Proceedings of the 20th USENIX conference on Security*, SEC'11, 2011.

[5] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner. Android permissions demystified. In *Proceedings of CCS 2011*, 2011.

[6] A. P. Felt, S. Hanna, E. Chin, H. J. Wang, and E. Moshchuk. Permission re-delegation: Attacks and defenses. In *In 20th Usenix Security Symposium*, 2011.

[7] Google. Monkeyrunner. http://developer.android.com/tools/help/monkeyrunner_concepts.html.

[8] M. Grace, Y. Zhou, Z. Wang, and X. Jiang. Systematic detection of capability leaks in stock Android smartphones. In *Proceedings of NDSS 2012*, Feb. 2012.

[9] M. Grace, Y. Zhou, Q. Zhang, S. Zou, and X. Jiang. RiskRanker: Scalable and Accurate Zero-day Android Malware Detection. In *Proceedings of the 10th international conference on Mobile systems, applications, and services*, 2012.

[10] M. C. Grace, W. Zhou, X. Jiang, and A.-R. Sadeghi. Unsafe exposure analysis of mobile in-app advertisements. In *Proceedings of WISEC 2012*, 2012.

[11] P. Hornyack, S. Han, J. Jung, S. Schechter, and D. Wetherall. These aren't the droids you're looking for: retrofitting android to protect data from imperious applications. In *Proceedings of CCS 2011*, 2011.

[12] Jesusfreke. smali. https://code.google.com/p/smali/.

[13] P. Kranenburg and D. Levin. Strace. http://sourceforge.net/projects/strace/.

[14] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Proceedings of PLDI '05*, 2005.

[15] A. Reina, A. Fattori, and L. Cavallaro. A system call-centric analysis and stimulation technique to automatically reconstruct android malware behaviors. In *Proceedings of EUROSEC 2013*, Prague, Czech Republic, 2013.

[16] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena. Bitblaze: A new approach to computer security via binary analysis. In *Proceedings of ICISS '08*. Springer-Verlag, 2008.

[17] Y. Tang, P. Ames, S. Bhamidipati, A. Bijlani, R. Geambasu, and N. Sarda. Cleanos: limiting mobile data exposure with idle eviction. In *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation*, OSDI'12, 2012.

[18] D. Wagner and D. Dean. Intrusion detection via static analysis. In *Proceedings of SP '01*, Washington, DC, USA, 2001. IEEE Computer Society.

[19] R. Xu, H. Saïdi, and R. Anderson. Aurasium: practical policy enforcement for android applications. In *Proceedings of USENIX Security 2012*, 2012.

[20] L. K. Yan and H. Yin. Droidscope: seamlessly reconstructing the os and dalvik semantic views for dynamic android malware analysis. In *Proceedings of USENIX Security'12*, 2012.

[21] C. Zheng, S. Zhu, S. Dai, G. Gu, X. Gong, X. Han, and W. Zou. Smartdroid: an automatic system for revealing ui-based trigger conditions in android applications. In *Proceedings of SPSM '12*, 2012.

[22] W. Zhou, Y. Zhou, X. Jiang, and P. Ning. DroidMOSS: Detecting Repackaged Smartphone Applications in Third-Party Android Marketplaces. In *Proceedings of the second ACM conference on Data and Application Security and Privacy*, 2012.

[23] Y. Zhou and X. Jiang. Systematic detection of capability leaks in stock android smartphones. In *Proceedings of NDSS 2013*, 2013.

[24] Y. Zhou, Z. Wang, W. Zhou, and X. Jiang. Hey, you, get off of my market: Detecting malicious apps in official and alternative android markets. In *Proceedings of NDSS 2012*, 2012.