

TRAPDROID: Bare-Metal Android Malware Behavior Analysis Framework

Halit Alptekin
Sabanci University

Istanbul, Turkey
halitalptekin@sabanciuniv.edu

Can Yildizli
PRODAFT

Yverdon-les-Bains, Switzerland
can@prodaft.com

Erkay Savas
Sabanci University

Istanbul, Turkey
erkays@sabanciuniv.edu

Albert Levi
Sabanci University

Istanbul, Turkey
levi@sabanciuniv.edu

Abstract—In the realm of mobile devices, malicious applications pose considerable threats to individuals, companies and governments. Cyber security researchers are in a constant race against malware developers and analyze their new methods to exploit them for better detection. In this paper, we present TRAPDROID, a dynamic malware analysis framework mostly focused on capturing unified behavior profiles of applications by analyzing them on physical devices in real-time. Our framework processes events, which are collected from system calls, binder communications, process stats, and hardware performance counters and combines them into a simple, yet meaningful behavior format. We evaluated our framework's detection rate and performance by analyzing an up-to-date malware dataset, which also contains specially crafted applications with malicious intent. The framework is easy to use, fast and providing high accuracy in malware detection with relatively low overhead.

Index Terms—mobile malware, dynamic analysis, android

I. INTRODUCTION

As Android malware analysis and detection systems evolve, attackers also adapt their methodologies by leveraging the progress in detection and analysis techniques. Modern malware samples are capable of detecting the presence of a virtualization environment or instrumented functions, which allows them to evade detection. Besides, they use techniques like obfuscation, encryption or dynamic loading to evade static analysis. They also use anti-emulation, anti-debugging techniques to evade dynamic analysis. There is a growing body of literature that recognizes the importance of detection of evasive malware [2]–[4].

In this paper, we present TRAPDROID, a scalable dynamic malware analysis framework that is able to analyze a given application in real-time on physical devices. TRAPDROID is divided into three components: *driver*, *agent* and *server*. The *driver* is a loadable kernel module which is responsible for interception of system calls, binder communications, collection of PMU (performance management unit) events, and statistical information of processes collected from kernel structures like `task_struct`. The raw events collected by the driver are securely transmitted to a user-level *agent* through custom character device. The *agent*, then, sends collected events to a remote *server* in a pre-processed format in order to reduce the traffic between the device and the server. All of the pre-processed events are populated into an advanced analysis engine using state-of-the-art machine learning techniques to

differentiate whether given application performed malicious activity or not. Our platform shows the results through a dashboard and allow users to involve in the analysis phase by triggering malicious activities of the target application.

TRAPDROID converts all pre-processed logs into unified behavior profile. This behavior profile format is simple yet adequate to represent the dynamic behavior of an application. The unified profile contains the following information:

- a selected subset of system calls such as `open`, `close`, `socket`, `connect` with returned value of Linux kernel;
- broadcast events such as `SMS_RECEIVED`, `SCREEN_ON`, `SCREEN_OFF` emitted by binder;
- IPC communication between target application and binder;
- PMU events of target process such as L1D-access, L1I-miss, instruction, branch-prediction count etc;
- accounting information of target process such as `maj_flt`, `min_flt`, `stime`, `utime`, `read_char`, `write_bytes`, etc.

In our experiments using the framework, we achieved 93.4% of accuracy on dataset which contains 281 malware and 355 benign applications collected from different research datasets. Moreover, we developed two sophisticated malware samples to analyze the behavior of different advanced malware techniques, such as cache-based side-channel attacks and covert channels.

First one is *ArmageddonApp*, which is a simple Android application equipped with ARMageddon [5] cache-attack library. *ArmageddonApp* targets the input library of the Android operating system and stores the attack results into a log file. Second one is the *Zpware*, a specially crafted Android application, which is able to collect network activities and accessible private data of the infected device and send them to a remote server through covert channel without requesting any internet permission from user. The results show that our framework is also capable of detecting these types of sophisticated malware behaviors.

The main contributions of our work are:

- A publicly accessible, open-source Android malware-analysis platform;
- Analyzing the differences between full-automatic and semi-automatic scan of evasive malware samples and em-

phasizing the importance of human factor in the malware analysis;

- Demonstrating that low-level features can be effectively used in Android malware detection, bridging the semantic gap between low-level features and malicious behavior;
- Emphasizing the importance of developing a defence against cache-attacks used by classical malware samples;
- Proposing several novel dynamic features and unified behavior profile to detect Android malware.

II. BACKGROUND

Android uses APK(Android Package Kit) as its application format. This format contains required files such as *AndroidManifest.xml*, icons, resources and executable *classes.dex* file which is compiled for DVM (Dalvik Virtual Machine). *AndroidManifest.xml* is the most important file among them since it contains many objects such as permissions, services, receivers, and providers. Android also contains a sandbox which prevents applications to interact between each others' data and processes directly. Data transfer between applications are handled by IPC calls through binder driver. Applications make use of *ioctl* system call in order to execute IPC calls of the Android operating system.

III. OVERVIEW OF FRAMEWORK

A. Architecture

TRAPDROID consists of three main components: *driver*, *agent* and *server*. While *driver* gathers the raw activity log from the kernel level, *agent* pre-processes collected logs and sends them to the remote *server*. As a next step, our server-side API converts logs into a simple, yet meaningful behavior profile. Figure 1 depicts the overview of our framework.

The *driver* module intercepts some selected system calls via altering system call table when mounted to the system. We have selected 29 of the most important system calls to reconstruct the behavior of an application. In addition, *driver* filters *ioctl* system calls on */dev/binder* to intercept 14 binder transactions. TRAPDROID is implemented in a modular way so it can be extended easily. Intercepted system calls and binder transactions can be seen in Table III.

Although LKM (Loadable Kernel Module) support is disabled by default in Android kernels, our *driver* cannot be inserted into already existing systems without re-compiling kernel. However, nowadays vendors give more opportunities to advanced users to hack and modify their own devices. Therefore, kernel re-compiling is not a major issue to use TRAPDROID.

The *driver* identifies whether the current process is our target or not by looking its *UID* value. This step is trivial as the Android operating system assigns a unique *UID* value to each installed application. When a system call occurs, the driver checks whether the current process' *UID* value is equal to our target *UID*. Then, the driver pushes a new event, which contains the information collected from *task_struct* and PMU, into a circular queue. This circular queue can be accessed from user-level via a character device. The *agent* starts the

different number of worker threads to pull a new event from driver's character device on */dev/trapdroid*. For the sake of simplicity, we opted to use character device rather than another transfer mechanism such as netlink. Due to the relatively small numbers of hijacked system calls, amount of produced logs is at minimum level as well.

The starting of Android applications can be captured by hijacking *fork* system calls of the *zygote* process. If *driver* detects a new application started in the operating system, it clears and enables the hardware performance counter of the target process for defined PMU events such as data cache access, branch-misprediction, instruction, etc. When the target application invokes the hooked system calls, *driver* stores the exact values of hardware performance counters. Targeting PMU events can be configured or extended easily.

strace is a debugging tool which is capable of intercepting all system calls invoked by the target process, taking advantage of the *ptrace* system call. However, evasive malware samples can identify whether their own processes are under analysis or not. Therefore, they can bypass the dynamic analysis systems based on *strace*. For instance, malware developers may craft a malware equipped with the native library, which is able to bypass *strace* based analysis for the Android operating system. As an another approach, the *perf* tool could be used to gather hardware or software events of the system. However, we preferred the unified behavior analysis approach rather than relying on different applications to gather different metrics. Moreover, in order to analyze a suspicious application, we need to be more privileged than the tracked application. Otherwise, evasive malware could bypass our analysis framework.

B. Bare-Metal Environment

Recently, researchers have shown an increased interest in automated dynamic malware analysis systems running on physical hardware. We have deployed our framework to a physical mobile device so as to conduct comprehensive analysis over real-world malware samples. Table I shows our analysis environment.

TABLE I
ANALYSIS ENVIRONMENT.

Model	Sony Xperia Z3 Compact
OS	Android 5.1.1
Chipset	Quad-core Krait 400
Memory	16 GB
Kernel	3.4-msm8974ac
Internet	Connected, no-restrictions
Cellular	Connected, without internet
PMU	Active, debug enabled

In order to start analysis process using physical hardware, partitions of the mobile device should be organized properly. Our framework keeps a clean state of the *system* and the *userdata* partition in a separate partition named *vault*. These partitions always keep the clean state of the device and are not mounted during the analysis.

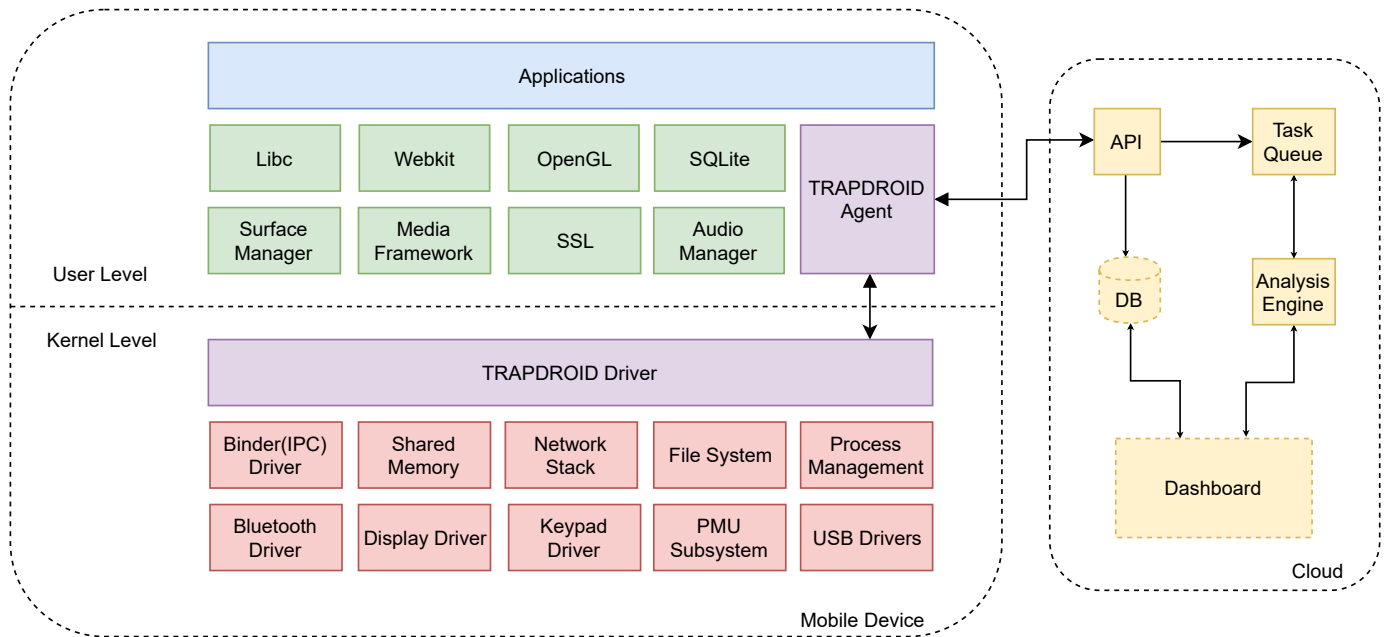


Fig. 1. Overview of TRAPDROID.

In case a modification is detected on the system partition, we use our previously saved state of the original system partition in our `vault` and flash it on the device accordingly.

We opted to use the threat model, which is explained in more detail in BARENDROID [1]. The applications under the analysis can read/write to the userdata partition. Additionally, we rely on SELinux policies to keep our system partition safe from malicious applications. We have a validation mechanism to determine whether the system partition is altered or not. TRAPDROID analyzes the past activities of malware and decides to upload a new partition into the device via ADB (Android Debug Bridge). After the analysis is completed, the userdata partition is always restored regardless of the malicious application's activity as in BARENDROID.

In the first phase of the restoration, TRAPDROID boots the device into fastboot mode. The content of the cache partition is erased in the fastboot mode. Then, the original boot partition is flashed, which contains the Linux kernel with our driver into the device. The device is booted into recovery mode after the first phase of restoring process.

In the recovery mode, we first change the contents of current userdata partition with a clean state of device byte by byte using the `dd` command. As a next step, we determine whether the system partition will be replaced or kept in the current state. TRAPDROID also uses checksum at this step to make sure whether the system and the userdata partition are kept intact. It also validates outputs of all commands executed for restoration during the analysis. In such cases, we also automatically push the original `vault` partition through ADB.

TRAPDROID uploads the agent into the system and execute it after ADB connection has established successfully. It waits the graphical user interface of Android operating system to

start the analysis process. After confirming every step during the preparation phase, we first start the main activity of application. Then we take the screenshot of main activity of sample under the analysis. As a next step, we execute monkey to send selected input events to application.

Moreover, we developed a simulator engine which uses monkey tool to mimic user actions and broadcast some important Android intent actions such as `BOOT_COMPLETED`, `SMS_RECEIVED`, `SCREEN_ON` to the application in order to trigger its malicious activity.

TABLE II
PROCEDURE TIMINGS.

Step	Time(seconds)
Restore the system partition	69.30
Restore the userdata partition	40.15
Boot the operating system	15.10
Flash the boot partition	0.75

We take advantage of the static analysis so as to uncover hidden behaviors of target application and reduce the analysis time. We have focused on neither how static analysis useful at malware detection nor to improve the static analysis techniques. We only leverage that static analysis report to improve the efficiency of dynamic analysis. TRAPDROID extracts the statically defined broadcast receivers, which are expected by the target application from its `manifest` file. In addition to static receivers, our framework also finds the receivers which are registered dynamically. Our simulator engine uses this knowledge to broadcast more relevant intent actions in order to interact with the application.

In our experience, the malicious behavior of the samples can mostly be triggered by our simulator engine.

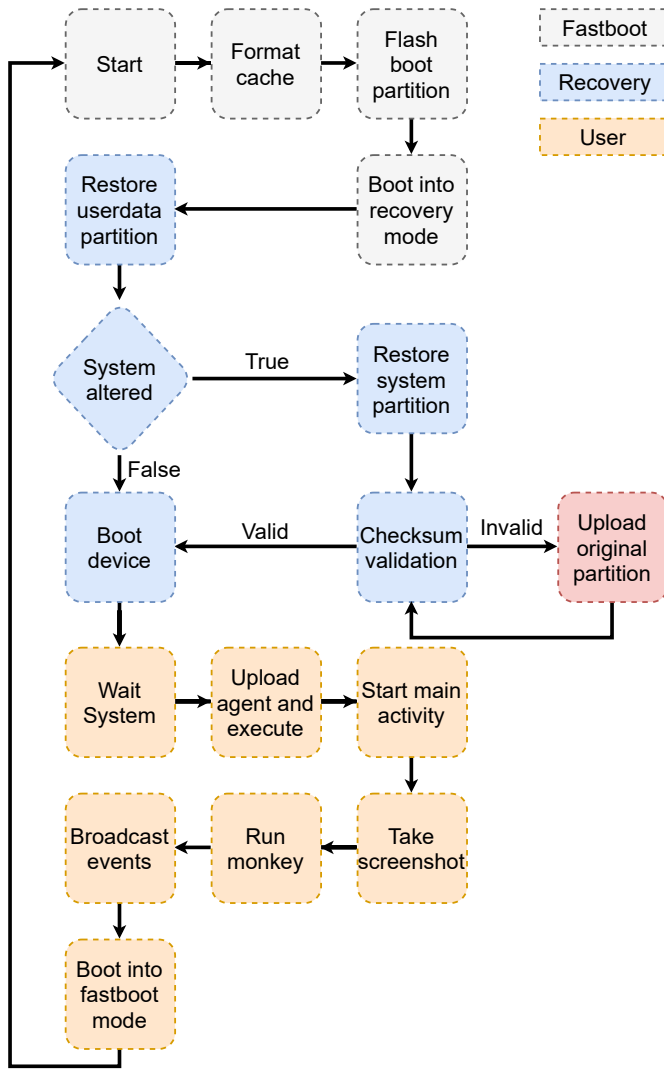


Fig. 2. Analysis procedure.

However, further investigation and experimentation of how to effectively trigger the malicious behavior of an application is another research area which needs to be addressed. Figure 2 depicts the analysis procedure which we apply to the given samples.

C. Unified Behavioral Profile

TRAPDROID converts pre-processed logs into simple yet powerful behavior profile format. We abstract the system calls, binder transactions and broadcast actions to model the behavior of applications during our analysis..

We map system calls which operate in same manner into same behavior. For example, if a process uses `open` or `openat` system call to open file. We map this behavior into `OpenFile`.

Kernel returns a handler, which represents operating system objects like `file`, `socket`, `thread` or `process` after the system call is performed. For instance, `open` system call returns with file descriptor. The analysis engine is capable

of resolving dependencies between system calls with tracking these handlers. In order to illustrate this as an example, the process first opens a file, invokes `ioctl` call and closes it. We are able to represent the flow between these calls using file descriptor object which is returned from the first `open` system call. Therefore, we can make better distinctions between the calls like: closing the socket or closing the file.

We also take advantage of system call arguments in order to produce meaningful representation of the application's behavior. For example, if the process opens a file with `O_RDONLY` flag, it becomes clear that the process wants only read permissions and will not expected to alter or delete the file. On the contrary, if the process opens the same file with `O_APPEND` flag, this means that the process wants to append data into the file. These represents different intents. This small but important detail allows us to learn more information about a system call and it enables us to make better reasoning about the process than other techniques in the literature. We map the arguments of system calls into meaningful attributes like `TCP_CONNECTION`, `HTTP_CONNECTION`, `READ_ONLY`, `WRITE_ONLY`, etc.

In addition to system call arguments, we also transform the return values of the system calls into meaningful representation. The return values of the system calls depend on the type of the system call. As an example, kernel returns `fd` that represents the opened file; whereas, it returns `pid` that represents the child process. In addition to that, we also make use of the error-codes from the return values of the system calls. If a process attempts to open a file that does not exist in the file system, kernel returns `ENOENT` error code to the user. We have a map table, which transforms the error codes like `ENOENT` to extended version of error codes like `NO_SUCH_FILE_OR_DIRECTORY`. Therefore, we can easily determine whether the process tries to open a file without required permission. This gives us signs about the behavior of a ransomware. Similarly, we represent the operating system object in compact form like `process[pid]`, `file[fd]`, `thread[pid]`. This representation allows us to determine and keep track of which thread or process is responsible for a system call or which operating system object like `file`, `socket` is affected by this system call.

IV. EXPERIMENTAL RESULTS AND FINDINGS

A. Novel Dynamic Features

Our dataset contains 281 benign and 355 malware samples which are obtained from [6], [7]. We applied different machine learning techniques in our framework to increase our detection rate for unknown samples. Our success rate reached to 93.4% after adjustments using *SVM* algorithm.

In order to test the output of our framework with the machine learning algorithms, we first converted all operations of the target application and their results into a feature vector. As an example, if any operation which results in a successful open operation for a given file, we encoded it as `OpenFile_SUCCESS`. Each encoded value then converted into a TF-IDF vector which became the input of the SVM

TABLE III
INTERCEPTED SYSTEM CALLS AND BINDER TRANSACTIONS WITH THEIR RESPECTIVE CATEGORIES.

Category	System Calls	Binder Transactions
Network	socket,connect	IWifiManager,IConnectivityManager,IBluetooth,IBluetoothManager
File System	open,openat,close,mount,umount,renamemat,symlinkat,unlinkat,mkdirat,faccess	IMountService
Process	fork,execve,clone,exit,sched_yield	IServiceManager, IApplicationThread
Injection	ptrace,uselib,init_module,delete_module	-
Privilege	setuid,setgid,setreuid,setregid,setresuid,setresgid,fchown,fchownat	-
Telephony	-	ITelephony,IActivityManager,IPackageManager,IPowerManager
Privacy	-	IPhoneSubInfo,ILocationManager, ISms

algorithm. Interestingly, we discovered that some of the malware we tested were actively trying to hide their presence by calling `setComponentEnabledSetting` method of `IPackageManager` class. We concluded that the observation of this function within an application effects the output of our framework drastically.

As another finding, we observed that there are significant differences between `stime` and `utime` values of a given application. Malware samples have a higher `stime` duration compared to their `utime` durations. Respectively, benign samples have a longer `utime` duration than `stime`. Additionally, `nvcsw`, `nivsw` and `cache` usages are other features, which can clearly be used in detection decisions. Difference between I/O operation rate with respect to broadcast events shows a significant indicator of malicious behavior.

TABLE IV
PERFORMANCE OF OUR CLASSIFIER.

	Precision	Recall	F1-Score
Malware	0.96	0.91	0.93
Benign	0.91	0.96	0.94

B. Advanced threat use-cases

It has been established that finding up-to-date malware samples is a very tedious process. Furthermore, most of the advanced attack scenarios, which are discussed in scientific researches, cannot be simulated due to the absence of practical samples. In our work, we developed different advanced malware samples that have novel attack capabilities. We demonstrated that our framework is also capable to detect future attack vectors which is likely to be seen in the practical world in the near future. This section explains multiple techniques that we have experimented during our research and how a specially crafted malware can be identified with `TRAPDROID`.

1) *ZPware*: Applications which are running on Android platform does not require any special permissions to check whether the screen is on or off. *Zpware* uses the weakness of this feature as an advantage to detect whenever a user is actively using the device. After detecting and active usage, malware reads all contents of the files under `/proc/net`

folder. These files contain metadata of all the network traffic of the device. Accessing the contents under this folder is clearly a major breach of the user's privacy.

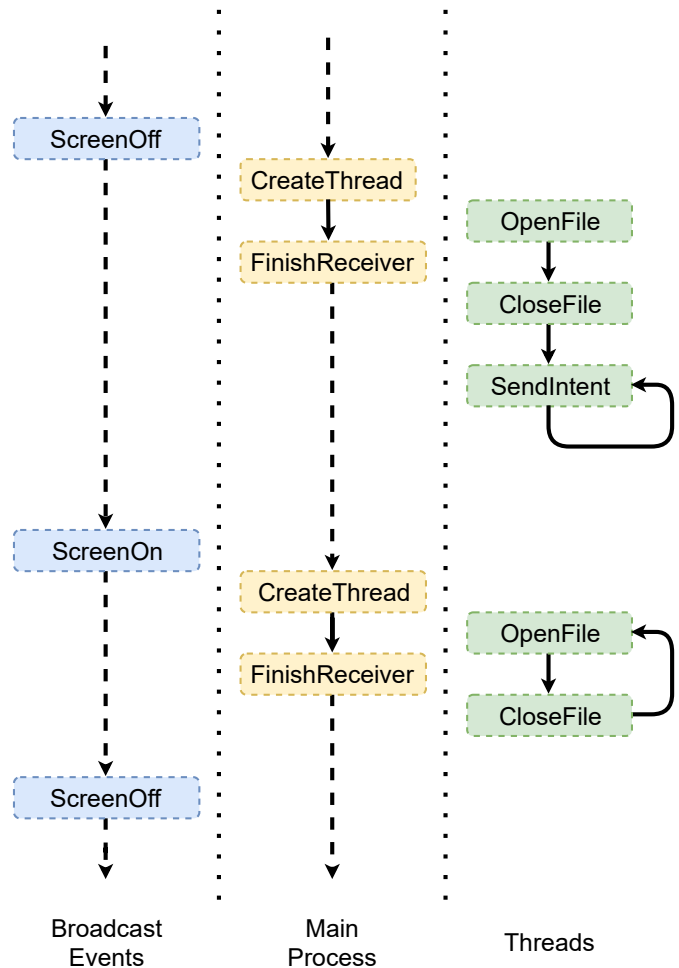


Fig. 3. Behavior activity graph of *Zpware*.

In addition to collecting network metadata, *Zpware* also processes the outputs of the tools such as `getprop` which contain more sensitive information such as device model, kernel version, etc. Whenever the user locks the screen or

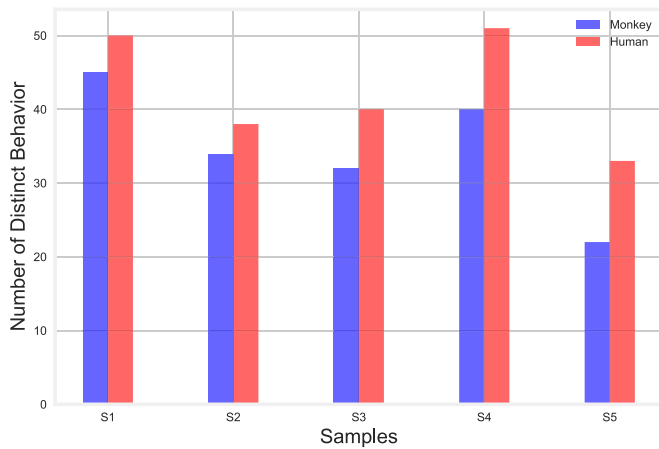


Fig. 4. Monkey vs Human

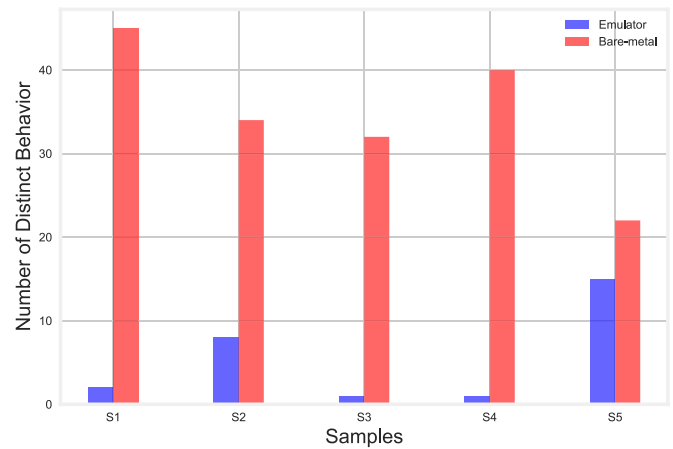


Fig. 5. Emulator vs Bare-metal

simply turns it off, malware automatically sends all collected information to a remote server using covert channels.

It is widely known that Android applications can create new `Intent.ACTION_VIEW` with a specially crafted URL argument and forces to open this URL with the default browser. In our example, we also make use of this technique to send collected data to a remote server in a covert fashion. There are few steps needed to successfully transfer large files since the GET requests that are called by the browser itself has a limited data header. Firstly, we need to divide the collected data into chunks. After that, we use encoding to transfer collected data chunks with the help of the `Intent.ACTION_VIEW` intent. Our sample uses the same browser page iteratively without open a new page for every intent.

We explained an advanced malware, which is very hard to detect, with static analysis since it does not require any permission and it does not contain any suspicious API calls. It should be noted that, the only feasible way to detect these types of advanced cases is through dynamic analysis. We used TRAPDROID to analyze *Zpware* and Figure 3 shows only a small part of its behavior. It is clear to see from this figure that the application immediately starts a new thread after the screen goes off. It should also be easy to observe that the number of file system activity is relatively very high when the screen goes back on. Our findings show the importance of analyzing events and their interactions with other applications in a broader picture to understand any malicious activity. Most of the malware has a trigger mechanism for their malicious actions such as receiving an SMS from the server, checking for the existence of a file and so forth. TRAPDROID automatically stimulates these actions and makes it very easy to analyze the behavior of these applications under different conditions.

2) *Cache Attacks on Mobile Devices*: Cache attacks are becoming more popular in malware field during the past few years. Many libraries and toolkits are released to public in order to practically experiment the outcomes. One of the famous tool, *ARMageddon*, is used to conduct cache attacks for ARM platform. We could not find any practical example of

a malware which uses cache attacks in the Android world so we developed one sample which uses the *ARMageddon* library to test our framework.

Our example carries a compiled *ARMageddon* library within itself and conducts cache-attacks to `libinput.so` library. It is relatively easy to capture all input activities of the user and save it into a file. After certain decision rules, the malware sends the collected data to a remote server like in the *Zpware* case. The main purpose of creating such a malware is not about testing the effectiveness of detecting cache attacks, but to show that our framework has the capability to detect possible practical cache attacks. In Figure 6, we represent some of the values of our *ArmageddonApp* sample during execution. One of the most important finding is the duration of the `stime` in applications which executes cache-attacks. Our reasoning is that the density of the `sched_yield` system calls which are used during the cache-attacks increases the duration of `stime` significantly. One can argue that `sched_yield` system call is not the only way to successfully execute a cache-attack. However, our research shows that a sample cache-attack scenario can also create deviations which allows our framework to detect it successfully.

C. Monkey vs Human

Most of the malware need some form of interaction for stimulation such as inputs from the user or occurrence of certain events. In order to test the efficiency of the proposed system, we have measured the number of triggered functions using the monkey tool and an expert analyst. In Figure 4, we show the number of distinct behavior actions of the malware which is triggered with the help of an expert analyst and the monkey tool. The results show that an expert analyst made it possible to observe 22.5% more actions on average than the automated counterpart. Triggering more functions by an expert analyst shows the advantages of using bare-metal environments for malware analysis as the number of observable functions differs significantly.

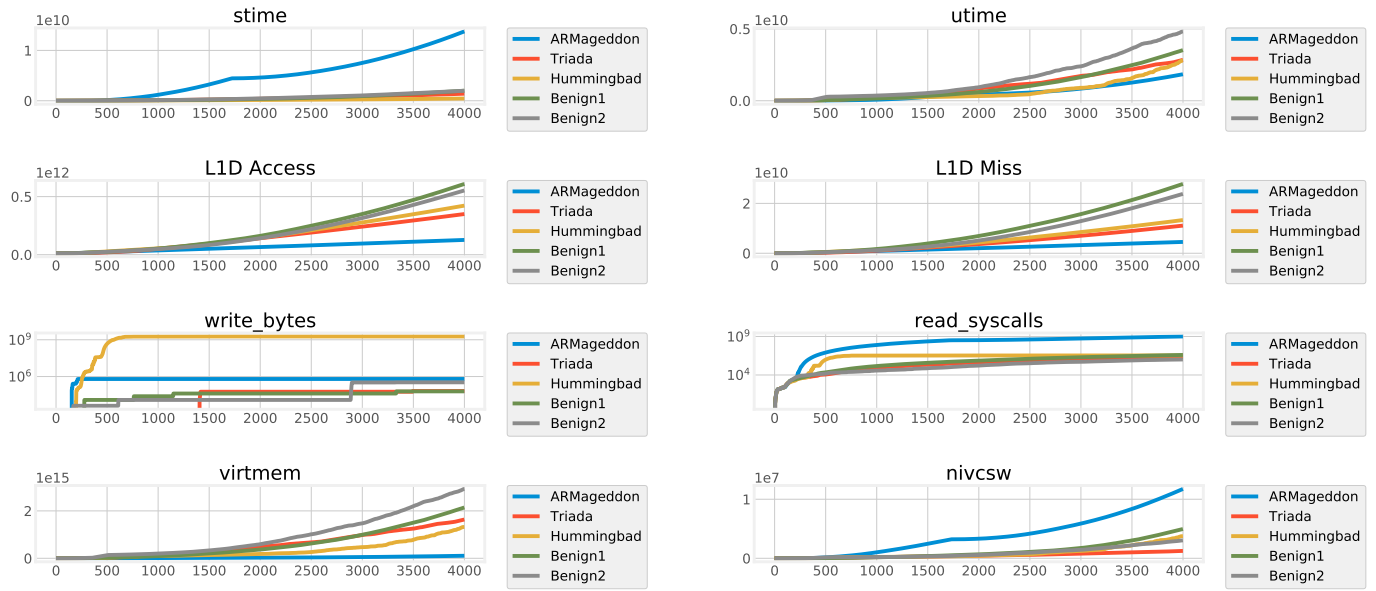


Fig. 6. Comparison of different metrics on various samples.

D. Emulator vs Bare-metal

In Section 4.3, we pointed out the execution difference between the emulator and bare-metal platform. We tested 5 different evasive malware samples which belong to *Ztorg*, *OBAD* and *Hehe* family. Our results can be seen in Figure 5. 4 of the evasive malware terminated upon execution in the emulator either because they detected the emulation or crashed due to failed state while the last of the 5 (S5) failed at a later step. All executions were successful in bare-metal framework which allowed us to analyze the whole malicious activity of the samples.

V. RELATED WORK

Crowdroid [8] and DroidTrace [9] uses ptrace to log and analyze application's system calls. However, it is fairly easy for an evasive malware to detect the presence of a debugger and ptrace monitoring. DroidScope [10] and CopperDroid [11] also make use of binder transactions in addition to system calls. Both of these tools based on QEMU and it does not represent the behavior of physical devices. BareDroid [8] uses the power of bare-metal platform. However, instead of using it for a practical analysis, authors explained the feasibility and procedures of running a bare-metal system for detection. In our work, we analyze the behavior of the applications on top of a bare-metal platform and create a unified profile in real-time. As far as we know, our framework is also the only example which combines PMU events, system calls, and binder transactions for analysis in a single framework.

VI. CONCLUSION

Analyzing malware in a bare-metal environment has a huge advantage against emulation. Malware continuously become more capable to detect and change execution flow in emulated environments. We present TRAPDROID as a practical example

of analyzing different malware types. We use unified behavior profiles instead of using a single system call metric. We also develop two advanced malware and analyzed their behavior difference in both emulator and our framework. Although our implementation shows many behavioral differences, more research is needed to understand system level properties such as cache usage and timing differences.

REFERENCES

- [1] MUTTI, Simone, et al. Baredroid: Large-scale analysis of android apps on real devices. In Proceedings of the 31st Annual Computer Security Applications Conference. ACM, 2015. p. 71-80.
- [2] BULAZEL, Web Alexei; YENER, Bülent. A Survey On Automated Dynamic Malware Analysis Evasion and Counter-Evasion. 2017.
- [3] JING, Yiming, et al. Morpheus: automatically generating heuristics to detect Android emulators. In Proceedings of the 30th Annual Computer Security Applications Conference. ACM, 2014. p. 216-225.
- [4] LIU, Lang, et al. RealDroid: Large-Scale Evasive Malware Detection on "Real Devices". In Computer Communication and Networks (ICCCN), 2017 26th International Conference on. IEEE, 2017. p. 1-8.
- [5] LIPP, Moritz, et al. ARMageddon: Cache Attacks on Mobile Devices. In USENIX Security Symposium. 2016. p. 549-564.
- [6] WEI, Fengguo, et al. Deep ground truth analysis of current android malware. In International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment. Springer, Cham, 2017. p. 252-276.
- [7] ARP, Daniel, et al. DREBIN: Effective and Explainable Detection of Android Malware in Your Pocket. En Ndss. 2014. p. 23-26.
- [8] BURGUERA, Iker; ZURUTUZA, Urko; NADJM-TEHRANI, Simin. Crowdroid: behavior-based malware detection system for android. In Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices. ACM, 2011. p. 15-26.
- [9] ZHENG, Min; SUN, Mingshen; LUI, John CS. DroidTrace: A ptrace based Android dynamic analysis system with forward execution capability. In Wireless Communications and Mobile Computing Conference (IWCMC), 2014 International. IEEE, 2014. p. 128-133.
- [10] YAN, Lok-Kwong; YIN, Heng. DroidScope: Seamlessly Reconstructing the OS and Dalvik Semantic Views for Dynamic Android Malware Analysis. In USENIX security symposium. 2012. p. 569-584.
- [11] TAM, Kimberly, et al. CopperDroid: Automatic Reconstruction of Android Malware Behaviors. In NDSS. 2015.



Halit Alptekin received the BS degree in computer engineering from the Karadeniz Technical University in 2011. He is currently a M.Sc. student at Sabanci University. He has been a partner of PRODAFT Cyber Intelligence, the clients of which were primarily financial institutions. His work at the PRODAFT included extensive cyber security experience both in malware and network security. He has been a speaker at various public and private cyber security conferences. A significant portion of Mr. Alptekin's time is spent on analyzing potential threats of mobile

malware and embedded systems. He also has been a radio amateur operator since 2005.



Can Yildizli received BS and MS degrees in Computer Science and Engineering from Sabanci University. He joined the Phd. program at Rensselaer Polytechnic Institute (RPI) in 2012 under the supervision of Prof. Bülent Yener. He is currently the Chairman of the Board of Directors and Chief Executive Officer of PRODAFT and INVICTUS companies. Mr.Yildizli co-authored various academic papers and journals related to malicious software and privacy projects. His research interests include Malicious Software and Privacy.



Erkay Savaş received the BS (1990) and MS (1994) degrees in electrical engineering from the Electronics and Communications Engineering Department at Istanbul Technical University. He completed the Ph.D. degree in the Department of Electrical and Computer Engineering (ECE) at Oregon State University in June 2000. He had worked for various companies and research institutions before he joined Sabanci University as an assistant professor in 2002. He is the director of the Cryptography and Information Security Group (CISec) of Sabanci University.

His research interests include cryptography, data and communication security, privacy in biometrics, trusted computing, security and privacy in data mining applications, embedded systems security, and distributed systems. He is a member of IEEE, ACM, the IEEE Computer Society, and the International Association of Cryptologic Research (IACR).



Albert Levi is a professor of Computer Science and Engineering in Sabanci University, Faculty of Engineering and Natural Sciences, Istanbul, Turkey. He received Ph.D. degree in Computer Engineering from Bogazici University, Istanbul, Turkey, in 1999. Previously, he served as a visiting faculty member in the Department of Electrical and Computer Engineering, Oregon State University and as a visiting professor in the Faculty of Computer Science, Dalhousie University. His research interests include computer and network security with emphasis on

mobile and wireless system security, public key infrastructures (PKI), privacy, and application layer security protocols. Dr. Levi has served in the program committees of various international conferences. He also served as general and program co-chair of ISCIS 2006, general chair of SecureComm 2008, technical program co-chair of NTMS 2009, publicity chair of GameSec 2010 and program co-chair of ISCIS 2011. He is editorial board member of The Computer Journal published by Oxford University Press and Computer Networks published by Elsevier.