Journal of Computer Security 28 (2020) 157–189
DOI 10.3233/JCS-191313
IOS Press

Identifying parasitic malware as outliers by code clustering

Hongcheng Li [a,b], Jianjun Huang [a,b,*], Bin Liang [a,b], Wenchang Shi [a,b], Yifang Wu [a,b] and Shilei Bai [a,b]

[a] *School of Information, Renmin University of China, Beijing, China*
[b] *Key Laboratory of DEKE, Renmin University of China, MOE, China*
E-mails: owenlee@ruc.edu.cn, hjj@ruc.edu.cn, liangb@ruc.edu.cn, wenchang@ruc.edu.cn, lizhiwuyi@ruc.edu.cn, baishilei@ruc.edu.cn

Abstract. Injecting malicious code into benign programs is popular in spreading malware. Unfortunately, for detection, the prior knowledge about the malware, e.g., the behavior or implementation patterns, isn't always available. Our observation shows that the logic of the host program is normally unclear to parasitic malware developers, resulting in very few interactions between the host and the payloads in lots of parasitic malware. Thus we can expose the injected part by grouping the code based on the interactive relations. Particularly, we partition a target program into modules, extract the relations, cluster the modules and further inspect the outliers to identify such malware. In this paper, we design a two-stage code clustering-based approach to detecting two representative types of malware, the UEFI rootkits and the piggybacked Android applications. Parasitic malware is reported when (1) any outlier in a UEFI firmware shows a relatively long distance to the largest cluster, or (2) the largest outlier distance exceeds zero in an Android application, i.e., multiple cluster exist after re-clustering outliers. We evaluate the approach on 35 pairs of benign/infected UEFI samples we do our best to get and achieve an overall *F1* score. of 100%. Applying the learned threshold to 50 other benign firmwares, we identify them without false positives. In addition, our evaluation on 1079 pairs of Android applications, shows an *F1* score of 90.66% when the third-party libraries are eliminated and a score of 87.36% if we keep the popular third-party libraries, demonstrating the effectiveness of the approach.

Keywords: Parasitic malware, outlier, code clustering, UEFI rootkit, piggybacked Android application

1. Introduction

One of the most popular ways to spread malware is to inject malicious code (*payload*) into benign programs (*host*), taking advantage of their popularity and achieving stealth to some extent. Such malware is known as the parasitic malware [33]. Studies have shown that such malware spreads in not only the high-level applications in the mobile and desktop platforms [38,86] but also the fundamental architectures, e.g., the computer firmware [50].

Great efforts have been taken to detect malware. Traditionally straightforward approaches try to recognize the software behaviors and check whether the behavior patterns match any known signatures [11,36,87]. Some other solutions extract the features from the embedded malicious code and apply mining-based methods to match similar code fragments from unknown programs, using the features as seeds [23]. Clone detection based approaches have also been presented when multiple instances of one

*Corresponding author. E-mail: hjj@ruc.edu.cn.

application are available [9,10,69,77], inspecting the differences among the instances. In Android platform, third-party libraries that are introduced by developers share some commonness with the injected payloads and researchers have devised techniques to identify such libraries [5,47,49,57]. Typically, third-party library detection either works similarly with the above mentioned malware detection by matching the collected signatures with any target applications [5] or looks for the commonly used code modules among millions of applications [47].

While all those approaches can be employed to detect malicious payloads within benign programs, it is notable that the *prior knowledge*, e.g., the implementation signatures and behavior patterns, is not always available. For example, we may lack of necessary signatures and thus miss an unknown type of malware, or we fail to determine a baseline as the seed, leading to massive false alarms. Besides, in realistic scenarios, obtaining multiple instances of a program or maintaining a corpus of millions of applications may not be applicable. All these aspects prevent us from finding a way to get a promising method to detect the parasitic malware. The challenge raises especially when we have only a small number of unlabeled applications but have to decide which contain the malicious payloads.

Though the parasitic malware can be in the form of advanced persistent threat (APT) that is carefully crafted with tightly entangled components, *our work targets such a type of parasitic malware that holds merely necessary interactions between the payloads and the host without interfering with each other*. Previous studies have shown many examples of such kind of malware. For example, Wei et al. found from massive repackaged malware that half of the malware families even isolate their payloads from the benign part in the code [78] and Tian et al. easily identified thousands of such malware in which the malicious components are generally independent of the host portion [70]. Although APT-alike parasitic malware generally attacks a specific target, we think most adversaries are more willing to spread their payloads to diverse targets while they usually do not essentially understand the logic of the hosts, due to code obfuscations or unnecessity of in-depth understanding, resulting in loose connections between the inject components and the original code. As we can see from the examples in Section 2.3 and Section 3.3, the host programs typically consists of a large group of tightly connected modules, whereas the connection between the host and the payloads can be easily broken. Hence, we can expose the injected code by grouping the host and the potential payloads separately based on the interactive relations in any given program. In particular, we can divide a target program into modules, extract the relations as features, cluster the modules and further inspect the outliers to identify parasitic malware. By this means, we leverage only the information contained within the target program to detect the existence of the payloads, without caring about the signatures or patterns of the malware or requiring the assistance of many other correlated programs.

To demonstrate the effectiveness of this idea, we choose to detect two representative types of parasitic malware, the UEFI rootkits at the fundamental architecture and the piggybacked Android applications at a very high-level of the software system. On one hand, in recent years, UEFI (short for Unified Extensible Firmware Interface) has been the mainstream firmware standard and equipped in almost all newly personal computers. Unsurprisingly, adversaries have shown interest in compromising a target machine by implanting malware directly into its UEFI, and in July 2015, a weapons-grade UEFI rootkit was found in a leaked dataset from Hacking Team [50]. Such rootkits, existing in the firmware, can survive from operating system re-installation and even hard disk replacement, causing extreme dangers. However, we have not found any systematic detection for them. On the other hand, Android, as the most popular mobile operating system [13], plays an essential role in our daily life. In the meantime, piggybacked applications have long been a troublesome security threat to the users, the developers and the application markets. While plenty of work has been proposed to detect such malware [51,65,93], as

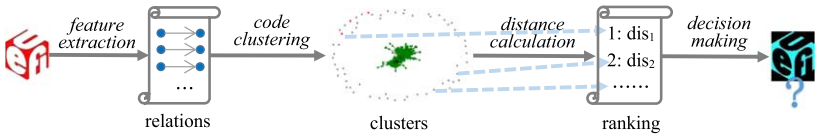


Fig. 1. Overall procedure of UEFI rootkit detection.

we mentioned previously, lack of prior knowledge can prevent us from effectively detecting piggybacked applications. We aim at devising lightweight techniques with the same core idea that work for the two different types of applications.

The integrity check could be a naive solution of defending the UEFI rootkits and piggybacked Android applications. For example, the Intel Boot Guard can be enabled by the motherboard manufacturers, measuring the integrity of the UEFI firmwares and refusing the execution of a parasitic firmware. However, boot guard mechanisms like the Intel Boot Guard have been reported to be bypassed by the adversaries. Moreover, the existence of the UEFI rootkits denotes either insufficient deployment or bypassed protection of the boot guard mechanisms, which motivated us to propose a new solution of the rootkits detection. Integrity checks can also be enforced by the Android system or the application developers. To conduct a system-level integrity check in the Android system, the system needs to *impossibly* maintain a large pool of the integrity information (i.e., the previously mentioned *prior knowledge*) for all legitimate applications, while such checks enforced within the application code can be bypassed by the adversaries during a repackaging phase.

In this paper, we design a two-stage code clustering based approach to identifying infected UEFI firmwares and piggybacked Android applications. Take the UEFI rootkit detection as an example. Figure 1 shows the high-level overview of the procedure. We extract the features (e.g., the GUIDs used in modules as depicted in Section 2.3) which can represent the interactive relations among modules, and based on the relations, group the modules into clusters. While the largest cluster contains the majority of the benign modules, all the payloads and some benign modules become left-out modules, a.k.a., outliers. We then computes the distance between each outlier and the largest cluster. If the greatest difference is higher than an empirical threshold, we consider the given UEFI firmware contains rootkits. Detecting piggybacked Android applications employs a similar code clustering stage, but leverages the organizational similarities to re-clustering the outliers, moving the left-out modules into bigger ones. Piggybacked applications are reported when eventually there does not exist a sole dominating cluster, i.e., multiple clusters exist and the greatest outlier distance is non-zero.

We implemented prototypes for both UEFI rootkit identification and piggybacked Android application detection, applied them to real-world parasitic malware and achieved good performance. Though a new implementation may be required for a new type of target, consisting of different preprocessing and feature extraction, we believe our work can provide guidance to future research for quickly discovering target specific features.

We make the following major contributions in this paper.

- We propose a code clustering-based technique to identify parasitic malware, in which the malicious code is exposed as outliers. We base the idea on the intuition that adversaries do not exactly know the logic of the host program and build few interactions between the payloads and the host. The approach leverages merely the given program to complete the detection, without requiring any prior knowledge about the malware or from other similar programs.

- We verify the effectiveness by detecting two representative types of parasitic malware, the UEFI rootkits and the piggybacked Android applications. For UEFI rootkit detection, after a clustering phase, we calculate the distance from each outlier to the largest cluster, which indicates an infected firmware if beyond a threshold. Re-clustering outliers based on the organizational similarities is applied for piggybacked application detection and more than one cluster at the end denotes the existence of payloads.
- We implement prototypes and evaluate their effectiveness on real-world malware. The experiments show that our approach achieves *F1* scores of 100% and 90.66%, individually for UEFI rootkit and piggybacked Android application (with third-party libraries eliminated) detection, which indicates a promising detection method for parasitic malware when prior knowledge is unavailable.

The rest of the paper is organized as follows. Section 2 presents the background information about the UEFI firmware and rootkits, as well as the detailed detection approach and the evaluation results. Section 3 adopts a similar structure as Section 2, but contains a case study section to show some interesting cases. We examine the limitations and some design decision in Section 4. Then we discuss the related work in Section 5 and conclude our paper in Section 6.

2. Detecting UEFI rootkits

In this section, we will detail our UEFI rootkit detection by first presenting the background information about the UEFI firmware and the rootkit, followed by the overview of our code clustering based detection method. After the overview, we will show the details of the approach and the empirical evaluation.

2.1. Background: UEFI firmware and UEFI rootkit

UEFI firmware is usually stored in some non-volatile storage devices, such as a flash chip or an EEPROM device [74]. The logical structure of the UEFI firmware is illustrated in Fig. 2. In brief, a UEFI firmware is composed of a series of firmware volumes, each volume contains a series of firmware files, and each file usually consists of several sections. Every component, i.e., the firmware, volume, file and section, contains a header, specifying information like the name, type and size. A file can be a driver, an application or simply a pad file that does not contain any code or meaningful data. The functionalities of a UEFI firmware are mainly achieved by various protocols, and the drivers and applications communicate with the protocols to fulfill their tasks [73]. Such modular design helps firmware developers and engineers better understand and debug the firmware, while at the same time, facilitates adversaries to write powerful firmware rootkits.

Compared with injecting and spreading rootkits to some high-level applications such as a Windows program or an Android application, infecting the firmware requires more efforts. Typically, the infection procedure consists of several steps. First, an adversary needs to reboot the target computer and enter the UEFI shell [63]. Second, he must dump the firmware out and implant the rootkit into the dumped file.

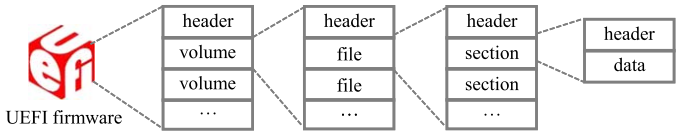


Fig. 2. The logical structure of the UEFI firmware.

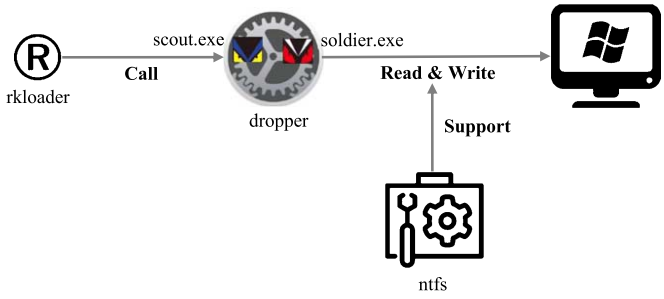


Fig. 3. The mechanism of Hacking Team's UEFI rootkit.

Finally, the infected firmware is re-flashed to the device and the computer system is restarted. Despite the infection difficulties, a UEFI rootkit can run long before the operating system starts and establish persistent and stealthy presence in the compromised machine, stealing private information or fully controlling the system without the user's awareness. No anti-virus tools can detect the rootkit as it is stored in flash chips. Even worse, re-installing the operating system or replacing the hard disk cannot remove the rootkit.

As an appealing research area, many researchers have devoted to study the firmware rootkit. Nevertheless, no really useful rootkit is released publicly until the Hacking Team data breach incident [81]. Some researchers found a UEFI rootkit in the leaked data and the rootkit could infect a number of computers [79,80]. Figure 3 illustrates the details of the Hacking Team's UEFI rootkit, which is comprised of two drivers (`rkloader` and `ntfs`) and one application (`dropper`). When the infected firmware is powered up, `rkloader` is loaded and triggers `dropper`. `dropper` utilizes the services provided by `ntfs` to touch the target file system. If the file system is writable and has not been infected before, `dropper` would try to implant two Windows Trojans (i.e., scout.exe and soldier.exe) into the target system. Every time the target machine boots up, the rootkit would perform the above operations.

To the best of our knowledge, no systematic detection of UEFI rootkits has been proposed yet. Though some researchers have proposed to correctly implement the UEFI Secure Boot mechanism to enhance the firmware security [8,82], such lightweight measurement can hardly be sufficiently enough. Boot guard mechanisms like the Intel Boot Guard intend to protect the firmwares via integrity check, but the existence of UEFI rootkits have denoted the deficiency of such protection. In this paper, we intend to fill the gap and design a generalized method for UEFI rootkits detection.

2.2. Approach overview

According to the modular design of UEFI firmwares, our intuition lies in two folds:

- Either malicious or benign functionalities are implemented in various modules, while the maliciously injected tend to behave differently from the benign ones.
- Adversaries do not exact know the logic in the host and thus few interactions occur between payload and host modules, but modules belonging to either category interact commonly to achieve their goals.

A straightforward idea is to group the modules, either host or payloads, into different clusters based on their intrinsic relations and then expose the outliers. Further inspecting the outliers can tell whether the target is infected. The interactions between modules can act as indicators for separating the modules.

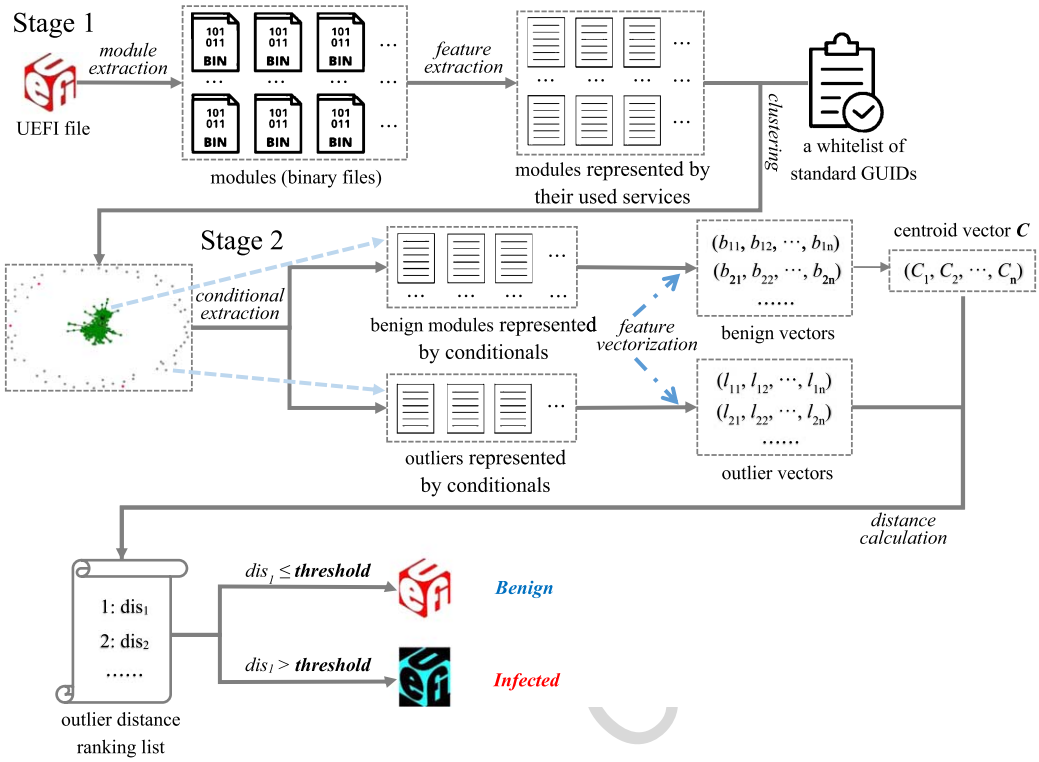


Fig. 4. Detailed procedure of UEFI rootkit detection.

The behaviors and interactions are represented within the code and can be acquired by static analysis techniques, and thereby we propose a code clustering based detection against the UEFI rootkits.

The approach consists of two stages, as shown in Fig. 4. During the first stage, we extract the modules, typically the binary files, from the given UEFI firmware, and then extract the features which represent the interactions between the modules and the underlying services. Using the filtered features to denote the relations among modules, we perform code clustering and most host modules (benign) are grouped into the largest cluster while out of the cluster are the potential rootkit modules and the ones (benign) without sufficient interactions with that cluster, majorly conforming our second intuition. The decision cannot be made at this moment because many benign modules become outliers as well. Then during the second stage, we apply a heuristic code analysis to characterize the module behaviors by their intrinsic conditional statements. We vectorize the characteristics for the modules, normalize the largest cluster as a centroid vector C and calculate the distances between outlier vectors and C. By this means, we are able to examine the top ranked ones for rootkits, and given a threshold of the distance, we can determine whether the given firmware is likely to be infected.

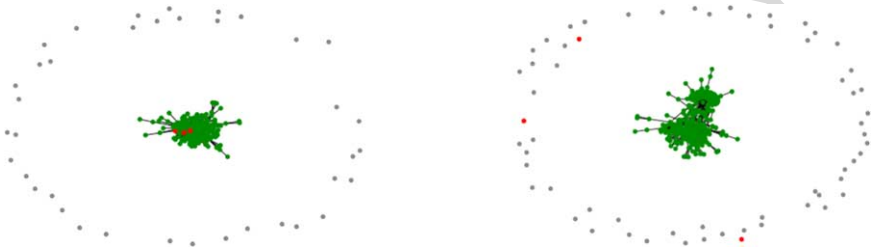
We will detail the two stages in the following sections and present our experimental results afterwards.

2.3. Code clustering

Since a UEFI protocol encapsulates a group of functionalities, the implementation and purpose can vary greatly from the others. The UEFI specification [73] claims multiple services for modules to register the protocol implementations and retrieve them for subsequent uses, or manipulate and access the data

Table 1
Eight UEFI services for module communications

Service name	Key parameters	Description
InstallProtocolInterface	EFI_GUID *Protocol	Installs a protocol interface on a device.
ReinstallProtocolInterface	EFI_GUID *Protocol	Reinstalls a protocol interface on a device.
InstallMultipleProtocolInterfaces	EFI_GUID **Protocol	Installs one or more protocol interfaces.
LocateProtocol	EFI_GUID *Protocol	Returns the first protocol that matches the given protocol.
HandleProtocol	EFI_GUID *Protocol	Queries a handle to determine if it supports a specified protocol.
OpenProtocol	EFI_GUID *Protocol	Open a protocol if it is supported by a handle.
SetVariable	CHAR16 *VariableName EFI_GUID *VendorGuid	Sets the value of a variable.
GetVariable	CHAR16 *VariableName EFI_GUID *VendorGuid	Gets the value of a variable.



(a) Clustering UEFI modules with all relations. (b) Clustering UEFI modules ignoring standard GUIDs.

Fig. 5. The clustering results of a UEFI firmware under different configurations.

stored in the protocols. Modules directly communicate with the protocols through such services and thus indirect communications happen among the modules. Then without considering how a protocol is implemented, it is reasonable for us to assume that, *two modules are correlated if they communicate with the same protocol*.

The official EDK II Module Writer's Guide [35] specifies eight commonly used services for modules to communicate with protocols. We present a simplified description of the services in Table 1. The first three services register one or more protocol interfaces, which can then be retrieved in modules by the three followed services. The last two services allow the modules to write and read specific information in the protocols. Each individual protocol interface is identified by a 128-bit globally unique identifier (GUID for short) and the data within a protocol can be accessed via the corresponding key name, a.k.a., the variable name in Table 1.

The GUIDs and variable names constitute a good feature, based on which we cluster the modules in one firmware. From each module, we recognize the aforementioned eight services and extract their key arguments. The concrete values indicate the connections between the modules. We group the ones with connections together. Figure 5(a) illustrates the initial clustering result for an infected UEFI firmware, in which the red dots denote the rootkit modules and the gray ones indicate the host modules that cannot be clustered into any groups. The majority, including the rootkit modules, form a huge cluster. Table 2 shows the extracted key argument values and the corresponding services in the three rootkit modules. We also label the values appearing more than once in the last column and it is easy to find that every two modules are indirectly connected by some protocols. For example, `ntfs` installs a protocol with

Table 2
The communication services used by the Hacking Team's UEFI rootkit

Module	Service	Key arguments	Label
rkloader	HandleProtocol	389f751f-1838-4388-83-90-cd-81-54-bd-27-f8	
	HandleProtocol	5b1b31a1-9562-11d2-8e-3f-00-a0-c9-69-72-3b	A
	HandleProtocol	09576e91-6d3f-11d2-8e-39-00-a0-c9-69-72-3b	B
dropper	LocateProtocol	0379be4e-d706-437d-b0-37-ed-b8-2f-b7-72-a4	C
	HandleProtocol	5b1b31a1-9562-11d2-8e-3f-00-a0-c9-69-72-3b	A
	HandleProtocol	09576e91-6d3f-11d2-8e-39-00-a0-c9-69-72-3b	B
	HandleProtocol	964e5b22-6459-11d2-8e-39-00-a0-c9-69-72-3b	D
	SetVariable	fTA, 8be4df61-93ca-11d2-aa-0d-00-e0-98-30-22-88	E
	GetVariable	fTA, 8be4df61-93ca-11d2-aa-0d-00-e0-98-30-22-88	E
ntfs	InstallMultipleProtocolInterfaces	18a031ab-b443-4d1a-a5-c0-0c-09-26-1e-9f-71	
		964e5b22-6459-11d2-8e-39-00-a0-c9-69-72-3b	D
	LocateProtocol	0379be4e-d706-437d-b0-37-ed-b8-2f-b7-72-a4	C
	HandleProtocol	5b1b31a1-9562-11d2-8e-3f-00-a0-c9-69-72-3b	A
	OpenProtocol	964e5b21-6459-11d2-8e-39-00-a0-c9-69-72-3b	
	OpenProtocol	964e5b22-6459-11d2-8e-39-00-a0-c9-69-72-3b	D
	OpenProtocol	ce345171-ba0b-11d2-8e-4f-00-a0-c9-69-72-3b	

label *D* which is retrieved by both `ntfs` and `dropper` via `OpenProtocol` and `HandleProtocol` respectively. All the three modules also query the corresponding handles to determine if the handles support the specified protocol labeled with *A*, which implies their intention of interacting with the same protocol.

While all the three rootkit modules are clustered together, unfortunately, as we can see in Fig. 5(a), they are tightly linked with the largest cluster, which should be recognized as the benign part. Our investigation shows that all the protocols listed in Table 2 are defined either in the UEFI specification or in the open source implementation of EDK II [71]. These *standard* GUIDs can be directly utilized by firmware developers to accelerate the development process as the corresponding protocols have been supported by the UEFI environment. For instance, the GUID *964e5b22-6459-11d2-8e-39-0-a0-c9-69-72-3b* labeled with *D* corresponds to the simple file system protocol which allows the code to obtain file based access to a device [73]. The uses of standard protocols in the original firmware modules and the rootkit make the huge cluster to enclose both benign and malicious.

Our solution to this problem is to exclude the standard GUIDs during the clustering phase with the help of a white list from [24]. The refined clustering generates a result shown in Fig. 5(b). The three rootkit modules in red are now disconnected with any other modules. The largest cluster contains only the original modules.

2.4. Rootkit detection

The preliminary experiments show that the white list aids code clustering in producing a huge cluster and a lot of outlier modules including the rootkit modules (Fig. 5(b)). While most benign modules are grouped together, we cannot directly distinguish the rootkit modules from the left-out host modules. Our intuition lies in that, *to keep the usability, the rootkit tends to probe the environment often to avoid exposing itself by an improper operation but the host modules are unlikely to do the same things*. In practice, an industrial or weapons-grade UEFI rootkit may face various platforms whose implementations differ largely. A simple UEFI firmware may only support minimal functionalities for the system to boot up and

```
lea ecx, ds:dword_A152B8
mov eax, [edi + 18h]
cmp eax, ecx
jz  near
```

(a) The branch statement with raw representation.

```
lea ecx, o_mem
mov eax, o_displ
cmp eax, ecx
jz  near
```

(b) The branch statement with normalized representation.

Fig. 6. A branch statement example in assembly language.

can only be accessed with a shabby user interface comprised of a white background and blue characters, but a well-designed firmware may look like a lightweight operating system with fancy functionalities such as Internet surfing. In order to adapt to varied environment, a rootkit usually avoids improper operations that may interfere with the infected system and then expose itself. Hence, the rootkit modules are likely to probe the environment to check whether certain properties are satisfied for specific operations, which illustrates different behaviors from the host modules. We are inspired to compute the differences between the outliers and the ones within the huge cluster, rank the outliers based on the differences and discover the most likely rootkit modules.

We concretize the probing as condition checks in the code, extract the branches from each module and characterize each module with the branches. In this paper, we limit a branch to four tightly correlated instructions, namely, a "jump", a "comparison" and two "assignment" instructions. The assignments define values that are used in the comparison and the comparison result determines where to jump. An example is shown in Fig. 6(a). After extracting all such branches inside all modules, we utilize the Bag-of-Words (BoW for short) [90] model to vectorize the modules. Specifically, each module is treated as a document and each branch instruction as a word. If a given UEFI firmware contains n different branch instructions, we sort the instructions in alphabetical order and represent each module with an n-dimension vector. The value of each entry in the vector indicates how many times the corresponding instruction occurs. For example, if a module contains only one branch as Fig. 6(a) shows, the vector would look like $(\ldots, 1, \ldots, 1, \ldots, 1, \ldots, 1, \ldots)$ in which the dots denote the absence of the other branch instructions in other modules. The vectors are then used to calculate the differences between modules.

Apparently, the raw instructions can introduce noises that affect the difference calculation. For example, two instructions "`lea ecx, ds:dword_ A152B8`" and "`lea ecx, ds:dword_A142D4`" perform semantically the same operations, fetching the value stored in the specified address and assign it to the `ecx` register. However, in the above naive approach, each instruction will occupy a position in the vectors, leading to different representation for the semantically same behaviors. In the simplest case, a firmware contains only two modules, one with the branch in Fig. 6(a) and the other with a slightly different `lea` instruction in the branch. Even though they are semantically the same, the corresponding vectors, (1, 1, 0, 1, 1) and (1, 1, 1, 0, 1) respectively, clearly show the differences. For a complicated firmware, the noises enlarge the effect and result in unusable difference values.

We adopt a normalization phase to address the issue. Simply, we use "`o_mem`" to represent the memory references (e.g., "`ds:dword_A152B8`"), "`o_phrase`" for the operands consisting of a base register and/or an index register (e.g., "`[edi+ecx]`"), "`o_displ`" for operands with one or more registers and a displacement value (e.g., "`[edi+18h]`") and "`o_imm`" for immediate values. We do not normalize any single register as each register has its unique application, for example, `eax` usually for storing the return value of a function call. The normalized form of Fig. 6(a) is shown in Fig. 6(b). By this means, the above simple case would generate the same vector (1, 1, 1, 1) for the two modules.

It is notable that we do not compute the difference between each outlier module and every module in the huge cluster. Instead, we treat the cluster as a whole and denote its vector as the centroid of the vectors for the enclosed modules. Specifically, the ith entry in the resultant n-dimension vector C is computed as below, where we assume the cluster contains m modules and $b[j]$ indicates the jth vector.

$$C[i] = \frac{\sum_{j=1}^{m} b[j][i]}{m}, \quad i = 1, 2, \ldots, n. \tag{1}$$

With the vectors, we calculate the difference between each outlier and C, and rank the results in a descending order. We can then inspect the top ranked to detect the rootkit modules which are more likely to behave differently with the benign modules. Moreover, the outlier with greatest difference can indicate whether the firmware is infected or not.

2.5. Evaluation

We leverage the IDA Pro [19] to generate the assembly instructions of a UEFI firmware and implement the code analyzer in a Python plugin. We use the plugin to collect the communication services, extract the key parameters such as the variable names and the GUIDs. We filter out the standard protocol GUIDs using the white list from [24] and group the modules based on the remaining GUIDs and variable names. After that, we apply another IDA plugin to extract the branch instructions in the modules and vectorize the normalized instructions before computing the differences between outliers and the known benign modules.

In this section, we first present the evaluation metrics and then talk about the dataset. Finally, we present the experimental results.

2.5.1. Evaluation metrics

A decent detection method should be able to detect malware with both low false positive rate and low false negative rate. A false positive means that the detection mistakenly identifies a benign sample as malware, while a false negative occurs when a malware sample is recognized as benign. To evaluate the effectiveness of the proposed method, we adopt the *recall* rate, *precision* rate and *F1* score to quantify the detection performance, which are calculated with the following equations. *TP* indicates the number of true positives, i.e., the number of malware which is detected as malware. *FN* denotes the number of false negatives and *FP* expresses the number of false positives. A high *F1* score represents good detection performance. Ideally, if the *F1* score is 1, it means the method can detect all the malware without any false positives or false negatives.

$$recall = \frac{TP}{TP + FN}, \qquad precision = \frac{TP}{TP + FP}, \qquad F1 = 2 * \frac{recall * precision}{recall + precision}. \tag{2}$$

2.5.2. Evaluation dataset

UEFI rootkit is a fascinating technology that has been studied by researchers from both academia and industry. However, to the best of our knowledge, no actually exploitable rootkit has been released except the Hacking Team's UEFI rootkit. Thus, we choose this rootkit as our detection target. We've done a lot investigations online, and find several computer models that can be implanted with the Hacking Team's rootkit [79,80]. The vulnerable models are Dell Precision T1600, Dell Latitude 6320, Asus X550C, Asus F550C and an unknown model. We've managed to implant the UEFI rootkit into the firmware of a Dell

Table 3
The basic information of the 35 benign experiment firmware samples

ID	Firm	Model	Version	#Module	ID	Firm	Model	Version	#Module
1	Dell	Precision T1600	A01	324	19	Dell	Latitude 6320	A08	351
2	Dell	Precision T1600	A04	328	20	Dell	Latitude 6320	A12	352
3	Dell	Precision T1600	A05	328	21	Dell	Latitude 6320	A13	352
4	Dell	Precision T1600	A07	331	22	Dell	Latitude 6320	A14	352
5	Dell	Precision T1600	A08	331	23	Dell	Latitude 6320	A15	352
6	Dell	Precision T1600	A10	331	24	Dell	Latitude 6320	A16	352
7	Dell	Precision T1600	A11	331	25	Dell	Latitude 6320	A17	351
8	Dell	Precision T1600	A13	331	26	Dell	Latitude 6320	A18	351
9	Dell	Precision T1600	A14	331	27	Dell	Latitude 6320	A19	351
10	Dell	Precision T1600	A15	330	28	Dell	Latitude 6320	A20	351
11	Dell	Precision T1600	A16	330	29	Dell	Latitude 6320	A21	351
12	Dell	Precision T1600	A17	330	30	Asus	X550C/F550C	205	139
13	Dell	Precision T1600	A20	330	31	Asus	X550C/F550C	212	139
14	Dell	Precision T1600	A21	330	32	Asus	X550C/F550C	217	139
15	Dell	Latitude 6320	A04	343	33	Asus	X550C/F550C	218	139
16	Dell	Latitude 6320	A05	348	34	Asus	X550C/F550C	300	139
17	Dell	Latitude 6320	A06	348	35	Unknown	Unknown	Unknown	219
18	Dell	Latitude 6320	A07	351					

Precision T1600 machine, and the rootkit works as expected. Every time the infected machine powers on, the rootkit will start and try to implant Trojans into the target system. The operations are performed very stealthily and can hardly be noticed.

Implanting the rootkit into real machines and then extracting the firmware code for detection can cost a lot and is really a waste. Fortunately, the infected firmware samples can be obtained through another way. We download benign firmware samples from the official websites, i.e., Dell [16] and Asus [4], and then manually insert the rootkit modules into the firmware samples with the help of some firmware editing tools, such as UEFITool [25]. In this way, we obtain 34 pairs of firmware samples for the experiments, each pair consisting of an infected sample and the corresponding benign sample. We also have one additional pair for an unknown model, the infected version of which is provided by the Hacking Team [80] and the benign part is manually extracted by removing the rootkit modules. The basic information of the 35 benign samples are presented in Table 3. The #Module column presents the number of modules in each firmware.

2.5.3. *Results and analysis*

Recall that the second stage ranks the outliers based on the differences between them and the centroid vector of the huge cluster. The top ranked modules are more likely to be the rootkit. Given the fact that our dataset is too small, instead of applying a cross-validation scheme to evaluate the effectiveness of our approach, we reasonably make the following assumption: *The legitimate UEFI firmwares for the same model but different versions should have very similar behaviors. Namely, the maximum distance from the top one outlier to the huge cluster for one version is very close to that for another version.* While the rootkit modules are expected to behave differently with the benign modules, we presume that the maximum distance from the top outlier to the centroid cluster in an infected firmware is at least multiple times of that in a legitimate version.

Table 4
Maximum distances of the three baseline sample pairs and the corresponding thresholds

ID	d_{benign}	d_{infected}	N	$\mathcal{T}$
9	730.36	5027.37	4	2921.44
27	728.85	5026.61	4	2915.40
31	496.86	5025.02	6	2981.16

Therefore, we evaluate our approach as follows. First, we randomly select three sample pairs from the three major models, Dell Precision T1600, Dell Latitude 6320 and Asus X550C/F550C, one pair for one model, and compute the largest distances d_{benign} and d_{infected} for each pair. Each d_{benign} is treated as the baseline d_{base} for the corresponding model. We set the threshold of the outlier distance as $\mathcal{T} = N \times d_{\text{base}}$, where a reasonable coefficient can be computed as $N = \lceil \frac{d_{\text{benign}} + d_{\text{infected}}}{2 \times d_{\text{base}}} \rceil$. Then, we cluster all the samples, and compute the maximum distance d_{test} for each firmware. At last, we compare each d_{test} with $\mathcal{T}$ for the same model and if $d_{\text{test}} > \mathcal{T}$, we report the corresponding firmware infected with the rootkit. Otherwise, we think the firmware does not contain any rootkit modules. We use the maximum threshold to test the unknown model (#35 in Table 3). By this means, we can inspect the precision and recall of our approach on the UEFI rootkit detection.

We present the maximum Euclidean distances and the corresponding thresholds for the three baseline sample pairs (#9, #27 and #31) in Table 4. When the three values are used to test the corresponding UEFI firmwares, we apply the highest value 2981.16 to the unknown model detection (#35).

Table 5 shows the final detection results for all samples and also the clustering statistics of the infected versions. The d_{test} column in the *Benign* column shows the maximum outlier distance of each benign sample. With the model-specific thresholds, we correctly identify all the targets without false warnings.

In the *Infected* column, the #Module column presents the total number of modules within each firmware, including the rootkit modules. The %Huge column shows how many modules are grouped into the largest cluster. From this column, we see that for all the Dell models and the unknown model, more than 80% of the modules finally belong to the huge cluster in each sample, while for the ASUS models, the percentages are greater than 79%. We further list the distances from the three rootkit modules to the corresponding largest cluster in each infected firmware. The maximum distance appear to be the distance from the `ntfs` module to the huge cluster. Since $d_{\text{test}} > \mathcal{T}$ is satisfied for all the 35 infected samples, we successfully report them to contain rootkit modules.

Eventually, we get zero false positives and zero false negatives, indicating that both the precision and the recall are 100% for our approach to detect the UEFI rootkits, resulting in a maximum *F1* score of one.

Despite the good detection performance, we must also notice that, not all the three rootkit modules have the distances greater than the threshold. In fact, the distances for the `rkloader` and `dropper` modules are much smaller than the baseline distances as shown in Table 5. The infected versions are reported because the `ntfs` module shows a substantial difference with all other host modules. As presented in Fig. 3, the `rkloader` driver only loads the `dropper` application into memory, and `dropper` leverages `ntfs` to implant Trojans. The major probing operations and core functionalities, such as checking whether the target system is writable and implanting Trojans into the target system, are handled by `ntfs`. Such differences can be observed from the assembly code. Moreover, `rkloader` contains only 197 instructions with 6 branches, and `dropper` contains 1144 instructions with 111 branches, while `ntfs` consists of up to 51,539 instructions with 7420 branches. Among all the benign

Table 5
The UEFI firmware clustering and rootkit detection results. "✓" in the detection column indicates a correct identification

ID	Threshold	Benign		Infected					
		d_{test}	Detection	#Module	%Huge	d_{rkloader}	d_{dropper}	$d_{\text{ntfs}}(d_{\text{test}})$	Detection
1	2921.44	729.62	✓	327	81.35%	11.33	60.43	5027.47	✓
2	2921.44	729.56	✓	331	81.27%	11.36	60.37	5027.44	✓
3	2921.44	729.54	✓	331	81.27%	11.37	60.35	5027.42	✓
4	2921.44	729.53	✓	334	81.14%	11.38	60.35	5027.41	✓
5	2921.44	729.53	✓	334	81.14%	11.38	60.35	5027.41	✓
6	2921.44	729.89	✓	334	81.14%	11.37	60.34	5027.41	✓
7	2921.44	729.88	✓	334	81.14%	11.39	60.33	5027.40	✓
8	2921.44	730.36	✓	334	81.14%	9.82	60.10	5027.37	✓
9	2921.44	730.36	✓	334	81.14%	9.82	60.10	5027.37	✓
10	2921.44	729.08	✓	333	80.78%	11.83	59.72	5026.77	✓
11	2921.44	729.08	✓	333	80.78%	11.83	59.72	5026.77	✓
12	2921.44	729.08	✓	333	80.78%	11.83	59.72	5026.77	✓
13	2921.44	729.08	✓	333	80.78%	11.83	59.72	5026.77	✓
14	2921.44	729.08	✓	333	80.78%	11.83	59.72	5026.77	✓
15	2915.40	729.37	✓	346	84.10%	11.46	60.21	5027.29	✓
16	2915.40	729.34	✓	351	84.05%	11.47	60.17	5027.28	✓
17	2915.40	729.32	✓	351	84.05%	11.48	60.16	5027.26	✓
18	2915.40	729.32	✓	354	83.90%	11.48	60.16	5027.26	✓
19	2915.40	729.30	✓	354	83.90%	11.50	60.15	5027.25	✓
20	2915.40	729.64	✓	355	83.66%	11.51	60.12	5027.23	✓
21	2915.40	728.87	✓	355	83.66%	9.99	59.91	5027.21	✓
22	2915.40	730.12	✓	355	83.66%	9.99	59.91	5027.21	✓
23	2915.40	730.12	✓	355	83.66%	9.99	59.90	5027.21	✓
24	2915.40	730.12	✓	355	83.66%	9.99	59.90	5027.21	✓
25	2915.40	728.86	✓	354	83.33%	11.96	59.53	5026.61	✓
26	2915.40	728.86	✓	354	83.33%	11.97	59.53	5026.61	✓
27	2915.40	728.85	✓	354	83.33%	11.97	59.52	5026.61	✓
28	2915.40	728.85	✓	354	83.33%	11.97	59.52	5026.61	✓
29	2915.40	728.85	✓	354	83.33%	11.97	59.52	5026.61	✓
30	2981.16	496.87	✓	142	79.58%	56.65	72.51	5025.03	✓
31	2981.16	496.86	✓	142	79.58%	56.71	72.54	5025.02	✓
32	2981.16	496.83	✓	142	79.58%	56.60	72.46	5025.00	✓
33	2981.16	496.83	✓	142	79.58%	56.60	72.46	5025.00	✓
34	2981.16	496.83	✓	142	79.58%	56.60	72.46	5025.00	✓
35	2981.16	728.75	✓	222	85.59%	92.58	107.88	5026.83	✓

test modules, the average and median number of branches are 282 and 57, respectively. The heterogeneity between `ntfs` and the host modules may account for the huge difference we observe.

2.5.4. *Results on more firmwares*

Due to the fact that we do not have many UEFI rootkit samples, we have no way to verify the effectiveness of the approach on other infected firmwares. However, we are able to test if it is capable to correctly identify the other benign UEFI firmwares not listed in the Hacking Team's dataset [50] with the above thresholds. To this end, we download 50 UEFI firmwares from different manufacturers, models

Table 6
Additional UEFI firmwares and detection results with the threshold as 2981.16. "✓" in the result column indicates a correct identification

ID	UEFI information	d_{test}	Result	ID	UEFI information	d_{test}	Result
1	Acer Aspire E5-771 1.23-v1	221.45	✓	26	ASUS TUF B450-PLUS Gaming 0402	445.58	✓
2	Acer Aspire E5-771 1.23-v2	367.91	✓	27	ASUS VivoBook 17 M705BA 300	457.18	✓
3	Acer Aspire TC-703 P11.B4	948.48	✓	28	ASUS VivoBook Flip TP501UQ 301	453.28	✓
4	Acer Extensa 2511 1.31	367.26	✓	29	ASUS ZenBook UX410UA 200	453.28	✓
5	Dell Alienware X51 R2 A04	419.49	✓	30	Gigabyte Aorus X3 Plus v6 04 v1	1912.68	✓
6	Dell Inspiron 5521_A14	221.77	✓	31	Gigabyte Aorus X3 Plus v6 04 v2	1912.68	✓
7	Dell Optiplex 9010 A22	1098.01	✓	32	Gigabyte Aorus X7 DT FB03FD03 v1	370.03	✓
8	Gigabyte Aero 14 FB07	1911.81	✓	33	Gigabyte Aorus X7 DT FB03FD03 v2	370.03	✓
9	HP 245 G3 Notebook F.45 v1	874.61	✓	34	HP ENVY 17-j000 Leap Motion F69 v1	519.14	✓
10	HP 245 G3 Notebook F.45 v2	1010.64	✓	35	HP ENVY 17-j000 Leap Motion F69 v2	220.67	✓
11	HP ENVY 17 k200 F57 v1	220.52	✓	36	Lenovo Flex 3-1130 C8CN31WW	367.08	✓
12	HP ENVY 17 k200 F57 v2	366.66	✓	37	Lenovo IdeaPad 110-17ISK 4RCN11WW	428.78	✓
13	HP ENVY 17 k200 F57 v3	366.66	✓	38	Lenovo IdeaPad 305-15IBD A8CN54WW	487.46	✓
14	HP ENVY 17 k200 F57 v4	368.01	✓	39	Lenovo ideaPad Yoga 530-14IKB 7QCN42WW	377.09	✓
15	HP ENVY 17 k200 F57 v5	220.52	✓	40	Lenovo IdeaPad Yoga 720-13IKB-80X6 2.07	363.13	✓
16	Lenovo ThinkPad E555 1.23	1006.01	✓	41	Lenovo V110-14AST 1PCN63WW	1903.99	✓
17	Lenovo ThinkPad E565 1.18	1043.39	✓	42	Lenovo Yoga 530-14ARR 8MCN52WW	850.71	✓
18	Lenovo ThinkPad E575 1.17	1037.33	✓	43	MSI GT73VR 6RF Titan Pro E17A1IMS.106	369.95	✓
19	Lenovo ThinkPad P70 1.15	1450.95	✓	44	MSI MPG Z390 Gaming Pro Carbon AC 1.1	1560.44	✓
20	Lenovo ThinkPad T460s 1.06	673.63	✓	45	Sony VAIO SVE14A15FXW R0300V4	933.99	✓
21	Lenovo ThinkPad T560 1.06	673.63	✓	46	Sony VAIO SVF1532DCXW R1100DB	480.92	✓
22	Lenovo ThinkPad X270 1.09	672.74	✓	47	Sony VAIO SVF15A1ACXB R0250DA	221.29	✓
23	Lenovo ThinkPad Yoga 15 1.20	319.72	✓	48	Sony VAIO SVT11223CXW R1170V8	221.87	✓
24	MSI MPG Z390 Gaming Plus 1.1	1560.44	✓	49	Toshiba Satellite Pro C850-B 6.80 v1	223.79	✓
25	Toshiba Satellite L50DT-A 1.30	932.02	✓	50	Toshiba Satellite Pro C850-B 6.80 v2	915.77	✓

and versions and apply the maximum threshold 2981.16 to them as done for #35 in Table 5. The UEFI information and detection results are presented in Table 6. We can see the diversity of the firmwares. Not to say those for different manufacturers, even for the same model, different versions can result in different clustering results. For example, #50 has a relative larger distance than that for #49, while both are developed for the same model. With the threshold as 2981.16, all the firmwares are reported not to contain rootkits, showing a zero false positive rate.

3. Detecting piggybacked Android applications

As we have mentioned in Section 1, piggybacked Android applications have resulted in many security problems with the proliferation of the Android system. In this section, we first give a background introduction of the Android applications and the piggybacking and then talk about the overview of our approach. The details of the code clustering and piggybacking detection are then discussed, followed by the empirical evaluation of our approach on real-world applications. At the end of this section, we present two examples to show why false positives and false negatives may occur and how we can further solve those problems.

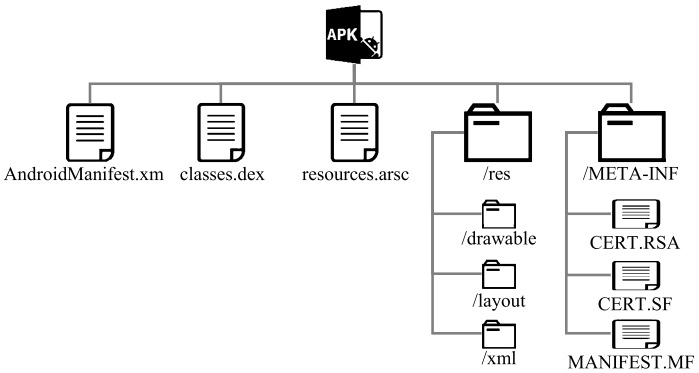


Fig. 7. The structure of a common APK file.

3.1. Background: Android application and piggybacking

An Android Package (APK) file is deployed by developers and installed as an Android application by the users. It is essentially a kind of ZIP file containing many directories and files, as Fig. 7 shows. `AndroidManifest.xml` declares the application name, version, requested permissions, components and some other application related information. `classes.dex` file contains the compiled Dalvik bytecode. Generally, an Android application implements the major functionalities in Java, which follows the design pattern of normal Java programs, dividing the code into packages, classes and methods [3]. For more details about the APK file, please refer to the online development documents [29].

In practice, with the help of reverse engineering tools, e.g., ApkTool [72], anyone can unpack an APK file, de-compile the code and get everything that constitutes the application. Then, injecting malicious code and redistributing the repackaged application can follow. Applications that are injected with malicious code are known as *piggybacked* applications. They may be injected/replaced with multiple Ad libraries for Ad revenues, or even steal confidential user information. From the dataset we described in Section 3.5.1, we find a pair of applications. One[1] is an utility tool, providing tutorials to help people keep fitness. The other[2] is the corresponding piggybacked version. We show the package structure of the piggybacked application in Fig. 8, in which the `fivefeiwo` part is the injected code and the `google` subtree corresponds to the third-party libraries. The core functionality of the fitness application is achieved by the remaining parts. The injected code launches a background service to perform stealthy operations, such as collecting the private information, e.g., the mobile number, the contacts and the SMS messages, and sending the information to a remote server.

As the widespread of Android applications, there is an urgent need for detecting the piggybacked applications effectively. Some researchers proposed clone detection-based techniques [9,10,69,77]. However, such approaches require both the original application and its piggybacked versions are in the same repository for detection, which apparently are not always applicable. Some others present matching-based solutions which match the applications features with known signatures [15,22,70], or monitor the runtime behaviors [67,85,88]. As we discussed in Section 1, such approaches heavily depend on the prior knowledge and can largely influence the detection performance with poor knowledge. We, in this paper, aim to propose an effective detection method that does not require any extrinsic information other than the given application itself.

[1] SHA-256 value: 03828A4CD0D2DCA3EC3441802FB35665AF42A894B5C794BE81A8873A3BE4FED7.
[2] SHA-256 value: 00B4C41891BC44B63A6A1C4B7ACA6B9B645B1B8579BAC2597D3503DD504F4612.

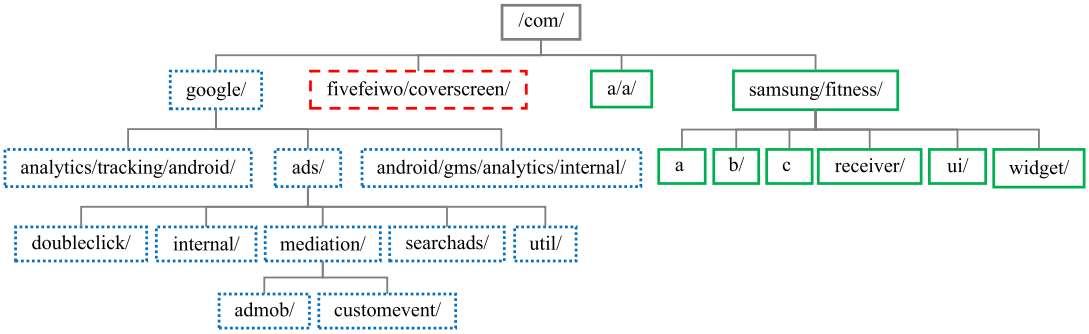


Fig. 8. The package structure of a piggybacked app.

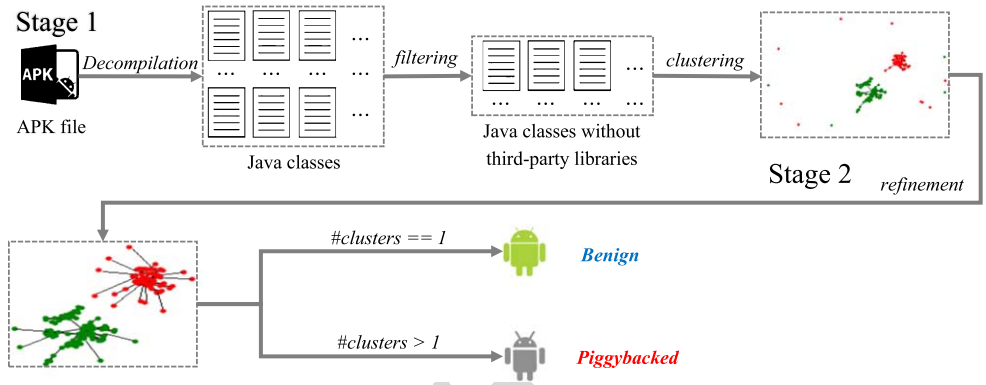


Fig. 9. Detailed procedure of piggybacking detection.

3.2. Approach overview

The fact that piggybacked Android applications work in common with the UEFI rootkit by implanting malicious payloads into benign applications and benefit from their wide spread motivates us to make the same assumption that the injected functionalities exist in modules and such modules are loosely connected with the host's code. Hence, we are inspired to design a similar code clustering based approach to detect the existence of piggybacking in Android applications. Considering the different implementation scheme, our approach treats each Java class as an individual module and groups the classes into different clusters after removing known third-party libraries. The outliers are further re-clustered based on their organizational similarities. If eventually two or more clusters exist, we consider the application piggybacked. In other words, we set the threshold of the maximum distance for outliers as zero and if the outlier's distance is larger than zero, the application is considered piggybacked. The two-stage workflow is shown in Fig. 9.

3.3. Code clustering

We treat a single Java class as a module and aim at clustering modules with strong interactive relations together. However, modules within Android applications do not interact with each other through GUIDs as what have been done in UEFI firmwares. Instead, they are connected by method calls, class inheritance, data access, etc. In this paper, we choose four types of interactive relations to feature the

connections between any two modules and exemplify the relations with the piggybacked Samsung fitness application mentioned in Section 3.1.

An is_a(C, S) relation indicates that one class C is a subclass of S. Class inheritance occurs commonly in Java programming as well as in Android programming. Typically we don't think the payloads will inherit many host classes when they are not specially designed for the target host application. This intuition makes the is_a relation suitable to weaken the connection between host classes and payload classes. The following code snippet presents an example in a piggybacked application. According to the Java grammar, we have is_a(CPManager, A), omitting the package name for brevity.

```
1 public final class CPManager extends A { .../* Details omitted */ }
```

has_a(C, F) tells whether a class C contains a field of type F. Enclosing an F-typed field enables C to access the full functions of F across all C's member methods with only one instance of F. In the simplified example below, has_a(MyApp, HomeKeyEventBroadCastReceiver) is satisfied.

```
1 public class MyApp {
2    private HomeKeyEventBroadCastReceiver c;
3    ... /* Details omitted */
4 }
```

In addition to the above relations, we take into account the data access via method invocations between modules. Method calls usually connect different code segments in a sequential manner and pass data bi-directionally via parameters and return values. It is reasonable to believe the application developers organize the functions into smaller modules (i.e., methods), chain them together and mostly transfer the data by method calls. Connections between methods further connect the enclosing classes. While inside the payloads method calls happen often, invocations from host modules to payload classes are assumed uncommon, let alone the data transmission between the host and the payloads. Since many method calls do not return data, we derive two relations to represent the effects of a method call concerning the data transmission. Figure 10 illustrates an example with the two relations, in which the left side shows the simplified code snippet and the right side presents how the relations are constructed on top of the data flows. The line numbers follow the '@' symbols.

Relation call_with(C, P, v_C) means within a class C, some method in P is invoked and some data v belonging to C is passed into the callee as an actual argument. We further reduce the relation to call_with(C, P), denoting that C invokes P's method somewhere. In Fig. 10, we say call_with(DetailActivity, S) satisfies because s.f(i) uses the local variable i at line 7.

```
1 package com.samsung.fitness.ui;
2 public class DetailActivity{
3    protected void onCreate(Bundle paramBundle){
4       int i = this.f.getInt("_id", 1);
5       ...
6       s = new com.samsung.fitness.a.S();
7       paramBundle = s.f(i);
8       this.d.setText(paramBundle.b());
9    }
10 }
```

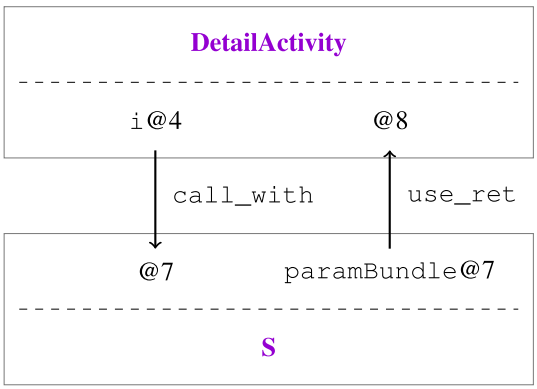


Fig. 10. Relations of call_with and use_ret in the simplified Samsung fitness application.

Note that we do not take the 'this' reference of an instance call into consideration, e.g., s in this example.

Relation use_ret(C, P) is used to represent that within some method of C, invoking a method of P returns some data which is further used in C. The return value is used if it appears in the right hand side of an assignment, the actual arguments of a method call, a return statement, etc. Similarly, in Fig. 10, use_ret(DetailActivity, S) holds.

To measure the strength of the connections, we think the amount of usages of another class C_b or relevant data in class C_a matters. For instance, if in C_a, a field of type C_b is used tens of times, the relation between C_a and C_b is much stronger than that between C_a and C_c when the field of type C_c is used only once. Similarly, if multiple method calls from C_a to C_b occur, it is natural to infer a close connection. In addition, for a single method call, heavy use of the actual argument or the return value represents a tight relation as well. We use $n(R, C_a, C_b)$ to express the amount of mutual usages. $n(\texttt{is\_a}, C_a, C_b)$ is always 1 or 0, depending on whether C_a is a subclass of C_b or vice versa. Below we will omit C_a and C_b for the sake of simplicity. $n(\texttt{has\_a})$ counts in C_a all uses of the fields typed with C_b and all uses of the fields of type C_a in C_b. For the other two types of relations, we count the accumulated uses of correlated values (i.e., the arguments or returns) for all methods calls between C_a and C_b. For instance, in Fig. 10, from what we have seen in the code, we have $n(\texttt{use\_ret}, \texttt{DetailActivity}, \texttt{S}) = 1$ and $n(\texttt{call\_with}, \texttt{DetailActivity}, \texttt{S}) = 1$.

While the above measurements are able to characterize the relations between classes, it is reasonable to assign different weights to them as we don't know which relations better describe the connection. For example, we think is_a denotes a much stronger relation than call_with because for adversaries it is easier to insert method calls in some host classes than to implement payload classes inheriting from host classes. We notate the weight with $w(R)$ where R is one of the four types of relations.

Considering both n and w, we compute the strength of relation between two classes as follows:

$$\mathcal{S}(C_a, C_b) = \sum\big(n(R, C_a, C_b) \times w(R)\big), \quad \text{where } R \in (\texttt{is\_a}, \texttt{has\_a}, \texttt{call\_with}, \texttt{use\_ret}). \tag{3}$$

The larger $\mathcal{S}$ is, the closer C_a and C_b are. However, we need a *threshold* to determine when $\mathcal{S}$ is large enough. Connections with $\mathcal{S}$ not larger than the threshold can be broken up. In this way, the modules are grouped into disconnected clusters. So far, we have five hyper parameters, including the four weights and one threshold. We will show later in the evaluation that different configurations can result in different clustering results. Take the piggybacked Samsung fitness application for example. If we assign the same weight to all types of relations and every connection is preserved with the threshold being zero, the payloads (red dots) are mostly connected with the host modules (green dots), as shown in Fig. 11(a). A better configuration will separate the payload cluster from the host cluster for the same application and the result is presented in Fig. 11(b).

3.3.1. Eliminating third-party libraries

It is notable that the third-party libraries in the applications are likely to cause false positives. Android application developers are fond of utilizing third-party libraries to accelerate the development, to enrich the functionality of the application, or to earn some Ad revenues. Take the original Samsung fitness application for example, it utilizes several third-party libraries. The *com.google.analytics* library help developers to better understand their customers by collecting and processing the network data that customers accessed, while the *com.google.android.gms* library provides services for users to use Google

(a) w(is_a)=1, w(has_a)=1, w(call_with)=1, w(use_ret)=1, threshold=0

(b) w(is_a)=5, w(has_a)=3, w(call_with)=1, w(use_ret)=1, threshold=4

Fig. 11. Clustering results of the piggybacked Samsung fitness application with different hyper parameter configurations.

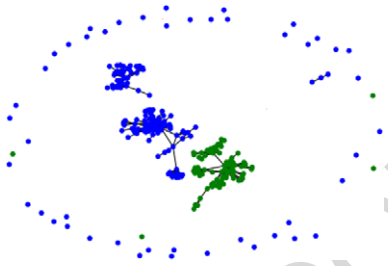


Fig. 12. The clustering result of the original Samsung fitness application containing third-party libraries (blue).

Search, Google Map, Gmail and other Google products. The *com.google.ads* package is an Ad library which can bring potential revenues to the developers. These libraries are of great help to developers, but can be pretty annoying when it comes to detecting piggybacked applications. In fact, none of the third-party libraries has many interactions with the core functionality of the application, which aim to providing tutorials on physical fitness. In other words, the third-party libraries possess similar relations to the host modules as the payloads, which introduce potential false positives. Figure 12 shows a clustering result of the original application under certain configuration. Most library modules in blue are clustered into two groups, demonstrating a similar result with the red payload modules in Fig. 11(b).

Our investigation shows that, a lot of studies have demonstrated the adverse effect of third-party libraries on malware detection and choose to remove the libraries [9,10,15,22,27,31,52,55,58,65,67,69,70,77,85,89,91–93]. We follow the same schema, identify and eliminate the third-party libraries from the target applications with existing tools before the clustering process.

3.4. Piggybacked application detection

After removing the third-party libraries and clustering the modules based on the aforementioned relations with certain configurations of weights and threshold, the results still cannot directly tell us whether a target application is piggybacked. As Fig. 11(b) and Fig. 12 show, not all green dots fit into the majorities, violating our assumption that benign modules should be clustered into one group. This phenomenon can be caused by dead code, limitations of the underlying analysis framework, etc. We skip deep program analysis and propose a lightweight solution. In this stage, we care about the organizational relations among the modules. According to the package organization property, we think classes under the same package have closer relation. Thus we try to re-cluster the outliers to generate fewer but larger clusters. In particular, for a left-out class *cls* belonging to a package *pkg*, if most (over 50%) classes within *pkg*

are clustered into a large group G, we move *cls* to G. If we cannot find such a cluster G, we leave *cls* as is.

Ideally, we have only one cluster for a benign application but at least two groups for a piggybacked one. We can then report the potentially piggybacked application if seeing multiple clusters.

3.5. Evaluation

We use the Soot framework [75] to analyze the Android applications and collect the relations, and utilize LibRadar [57] to identify the third-party libraries. To demonstrate the effectiveness, we apply the same evaluation metrics as in Section 2.5.1. We discuss below the dataset and our experimental results. The experiments are all conducted on a Dell laptop equipped with a four-core CPU and 8 GB memory.

3.5.1. Evaluation dataset

Though the Android application repackaging problem has been researched for years and a lot of work has been proposed to detect such malware, only a few work has released their experiment datasets publicly. Li et al. conducted massive experiments to collect original-repackaged application pairs [46,48]. An original-repackaged pair contains two applications, one repackaged and the corresponding original version. They have released a list containing all the pairs they collected, where each application is represented by its SHA-256 value and the applications are available from Androzoo [2], a growing repository of Android applications. We build our dataset on top of Li's work.

Since we focus on detecting malicious piggybacked applications, other types of repackaged applications are beyond the scope of this paper. However, Li's list contains various repackaged applications. To better evaluate the proposed approach, we exclude some pairs that do not meet our requirements. First, we exclude the pairs in which the original application is malicious or the repackaged is benign. It is fair to expect that natural applications coming from legal developers typically do not contain any malicious code, while piggybacked applications are usually meant to do something evil. Those pairs that do not fit into such expectation should not be taken into our dataset. By querying the VirusTotal website [76] with an application's SHA-256 value, we can get the feedback of whether the application is a malware. Second, we limit our method to detect piggybacked applications which are injected with some payloads. Therefore, we exclude the pairs in which the repackaged application only removes or modifies a few host code segments. We use ApkTool [72] to extract the classes from the application pairs. We consider the repackaged application to be piggybacked if it contains more classes than its counterpart. Finally, we obtain a dataset consisting of 1079 pairs. The APK file size ranges from a minimal value of 29.13 KB to a maximum value of 49.40 MB, with an average and median size of 11.13 MB and 9.20 MB, respectively.

3.5.2. Results and analysis

We evaluate our approach on the dataset. But first of all, the approach has four types of relations and five hyper parameters including the weights and threshold. We need to determine a suitable configuration of parameters and thus apply a cross validation to verify the effectiveness. Particularly, we divide the dataset into two parts, one for training and the other for testing. The training set contains 755 application pairs (70% of the dataset) and the testing set contains 324 pairs.

Arbitrary configurations may cost unlimited time to tune the parameters. To reduce the training cost, we empirically set small but suitable ranges for the parameters. In particular, the threshold ranges from 1 to 10 and the weights for `has_a`, `call_with` and `use_ret` range from 1 to 5. Considering that in Android application development, class heritage is a common feature to connect classes. Hence, we set w(`is_a`) as the value of the threshold plus one. We evaluate all the 1250 candidate parameter configurations on the training set, i.e., the 755 application pairs.

Table 7
Evaluation results for applications with the third-party libraries eliminated, corresponding to the highest, a top 4% and the lowest *F1* scores during the training phase

	w(is_a)	*w*(has_a)	*w*(call_with)	*w*(use_ret)	*threshold*	TP	FP	FN	Precision	Recall	F1
Training	**5**	**5**	**1**	**1**	**4**	**690**	**71**	**65**	**90.67%**	**91.39%**	**91.03%**
	5	5	1	5	4	678	67	77	91.01%	89.80%	90.40%
	11	1	1	1	10	724	208	34	77.68%	95.89%	85.83%
Testing	5	5	1	1	4	301	39	23	88.53%	92.90%	**90.66%**
	5	5	1	5	4	301	37	23	89.05%	92.90%	90.94%
	11	1	1	1	10	306	186	18	62.20%	94.44%	75.00%

Table 8
Evaluation results for applications without the third-party libraries eliminated, for the best parameter configuration during current training phase and the two selected in Table 7

	w(is_a)	*w*(has_a)	*w*(call_with)	*w*(use_ret)	*threshold*	TP	FP	FN	Precision	Recall	F1
Training	**5**	**5**	**1**	**5**	**4**	**694**	**310**	**61**	**69.12%**	**91.92%**	**78.91%**
Testing	5	5	1	5	4	311	77	13	80.15%	95.99%	**87.36%**
	5	5	1	1	4	312	86	12	78.39%	96.30%	86.43%
	11	1	1	1	10	316	278	8	53.20%	97.52%	68.85%

Table 7 presents the results for three parameter configurations, in which (5, 5, 1, 1, 4) and (11, 1, 1, 1, 10) represent the ones with highest and lowest *F1* scores during the training phase, respectively. The configuration (5, 5, 1, 5, 4) generates a top 4% *F1* score and is left for a subsequent comparison in Section 3.5.3. The top two configuration trades off the precision and recall with higher *F1* score, while the bottom one reduces false negatives (higher recall) but leads to more false positives (lower precision). In realistic scenarios, users can tune the parameters for either better precision or better recall, depending on their purposes.

We also present the testing results corresponding to the three parameter configurations in Table 7. The top configuration (5, 5, 1, 1, 4) results in a detection performance of a 90.66% *F1* score, with 301 TPs, 39 FPs and 23 FNs. Compared with that, the bottom configuration (11, 1, 1, 1, 10) leads to 145 more false positives but only 5 fewer false negatives, resulting in a low *F1* score of 75.00%.

3.5.3. The effect of third-party libraries

Due to the similarity between third-party libraries and the parasitic payloads, we eliminated the third-party libraries before the code clustering stage. In this section, we quantitatively show how the third-party libraries can affect the detection performance.

Without eliminating the third-party libraries, we conduct the same evaluation process on the dataset. In Table 8, we first present the parameter configuration that emits the best *F1* score in the training phase. We show the testing result of the configuration (5, 5, 1, 5, 4) and the corresponding precision, recall and *F1* score are 80.15%, 95.99% and 87.36%, respectively. The recall is relatively higher, compared with the precision, leading to an acceptable *F1* score. We also test the top and bottom configurations we present in Table 7. From the results, we can see that (5, 5, 1, 1, 4) continues to produce relatively high *F1* score while the other configuration results in low precision (53.20%) and *F1* score (68.85%).

When we further compare the results in Table 7 and Table 8, we observe that keeping the third-party libraries largely increase the number of false positives, more than 40 for top configurations and more than 100 for the bottom configurations. Considering the small number of testing application pairs, the precision rates are reduced by about 9%.

In contrast, fewer false negatives are reported if we do not eliminate the third-party libraries. That means, some adversaries inject the common third-party libraries as the payloads into legitimate applications and thus removing them from the detection can lead to false negatives. However, the side-effect of removing the third-party libraries is small. The recall rates show the decrement of only 3% from Table 8 to Table 7.

3.6. Case study

It can be noted that even the best configuration produces certain number of false positives and false negatives. In this section, we will present two typical examples.

The false positives are mainly caused by the loose connections among host classes. Take an original application[3] for example. The simplified code is shown in Fig. 13. As we can see, the class `LogEventReporter` is merely connected with `Config` via one simple `call_with` and one `use_ret` relation. The involved argument and return value are both used only once in `LogEventReporter`, resulting in the strength of relation $\mathcal{S}(\texttt{LogEventReporter}, \texttt{Config}) = 2$. However, our best configuration determines a threshold of 4, as shown in Table 7. Hence, the two classes will not be clustered together. Such kind of loose connections separate many host classes away from one major cluster in this application, generating too many small clusters which beyond the capacity of the second phase to build larger clusters. Thereby, a false positive emits when we see more than one cluster and report it piggybacked.

A potential solution to suppress such false warnings would be to take the indirect correlations into account. Currently, we only count the direct use of `obj` at line 6 for the `use_ret` relation. If we further consider the connection between `obj` and `json` at line 6 and impose a weighted value for the indirect uses of `json` and any further ones to the `use_ret` relation, we may probably avoid the false positive.

False negatives are produced because more connections than the threshold are found between host modules and the payloads. Figure 14 shows an example,[4] in which multiple connections from a host class `APPEntry` to a payload class `VManager` are constructed. The adversaries insert a field of type `VManager` into class `APPEntry`, which is used twice to call methods with the local values inside `APPEntry`. According to the best configuration, the strength of relation between two classes is at

```
1 package com.widdit;
2 import com.widdit.shell.Config;
3 public class LogEventReporter{
4    public void report(Context ctx){
5       Object obj = Config.getInstallDate(ctx);
6       json.put("installDate",obj);
7       ...
8    }
9 }
```

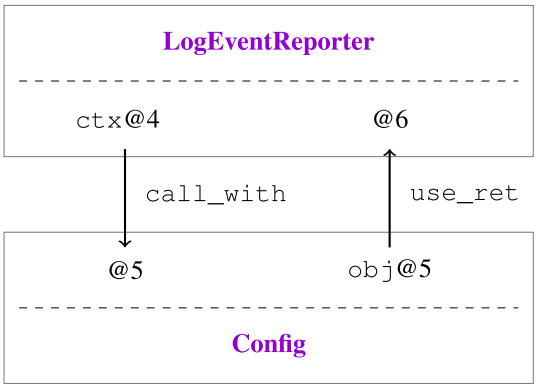


Fig. 13. A false positive caused by loose connections.

[3] SHA-256 value: 72C93B805A8DC118E2683C96F0AE67304D81999DC1EA4E215A54B58080D93E85.
[4] SHA-256 value: BE6D651F9BFFE1949D178A7B392E596CC9B949E140A797C287BE3BACDCABF561.

```
1  package air.com.ardman.sgreat.fullsteamahead;
2  import m11.a22.x33.VManager;
3  public class APPEntry{
4      private VManager vm = new VManager();
5      ...
6      public void InvokeWrapperOnCreate(){
7          String s = "2a98e5c787144aea9034ee3578b2";
8          Boolean b = new Boolean(true);
9          vm.setVID(s);
10         vm.getMessage(b);
11         ...
12     }
13 }
```

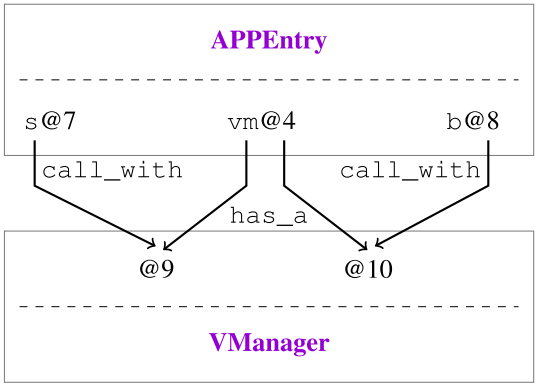


Fig. 14. A false negative caused by adversarially established connections.

least 12, based on only the code snippet in Fig. 14. While the strength threshold is only 4, two classes are grouped together and finally all payload classes are put into the sole cluster, leading to a false negative.

A possible way to evade the false negative is to apply a bottom configuration, e.g., (11, 1, 1, 1, 10) which will emit higher recall rate, according to Table 7. However, it obviously produces much more false warnings. We may also resort to some existing techniques, e.g., AsDroid [34], to identify the unexpected behaviors in such applications, so as to reduce the false negatives.

4. Discussion

The code clustering based parasitic malware detection approach has proven to produce acceptable performance for both UEFI rootkit detection and piggybacked Android application detection, with high *F1* scores. However, we have some additional concerns about our work.

The boot guard mechanisms like the Intel Boot Guard are supposed to be effectively protect the firmwares from being illegally modified. However, as discussed in Section 1, either the lack of the protection or the bypassed guards in real-world scenarios result in the existence of the UEFI rootkits. Our work aims at presenting a new detection approach to detecting such rootkits.

Our work focuses on detecting such a type of parasitic malware that introduces the malicious payloads as individual modules, so other types of repackaged malware, such as removing some code from the program or simply inject code deeply into the benign part (e.g., adding several lines of code in existing methods or inserting new methods into benign classes), are not handled. While such cases can happen in very few repackaged Android applications, we can draw support from some existing techniques to detect the malware. For instance, AsDroid [34] leverages the user interface information to determine whether the application behaves beyond the user's expectation and contains malicious code.

Following some previous studies that show loose coupling between the host code and the malicious portion in thousands of malware [70,78], we deem it popular for most adversaries to inject their payloads into host programs in such a way that leaves the two parts as independent as possible. We consider it reasonable as the software, either the UEFI firmware or Android applications, becomes complicated nowadays. For example, Android application developers tend to obfuscate the applications to protect them from being easily understood and modified. Take the popular instant messaging application WeChat v7.0.7 for example. 41,947 out of the 63,977 classes (64.86%) have obfuscated or meaningless names.

Considering the complication of the applications, it is very difficult, if not impossible, for adversaries to in-depth understand the internal logic of the host programs. Typically, we believe that the adversaries would like to quickly inject the payloads into as many targets as possible and thus they intend not to thoroughly analyze each application and disperse the payloads into different modules while keeping the functionality of the payloads. Even if the adversaries have the original code of a target application, it does not make any difference if the adversaries loosely compile the payloads together with the host program without a comprehensive understanding of the host. Moreover, we do not think it becomes a feasible task for the majority of adversaries to dive into the complex source code and seed the payloads in place.

We must admit that extreme cases can exist, in which the adversaries have the Pyrrhic knowledge of the host program and successfully spread the payloads in different modules. Our work will fail to detect such cases without the help of heavyweight prior knowledge of the malicious behaviors or the information of legitimate countermeasures. It is beyond the scope of our work, which aims at detecting the existence of parasitic components for any individual applications.

Adversaries may fake the interactions to bypass our machine learning based detection method, e.g., creating unnecessary field references and invoking meaningless methods. However, given the complication of the host programs, we believe the adversaries would take great effort to establish the extra connections between the payloads and the host for an individual application, making it unscalable to distribute the payloads in massive applications. Furthermore, we can evolve our solution to measure the quality of the interactions, e.g., the necessity of the interactions. Due to the lack of the cases, we cannot evaluate the effectiveness of our idea. In addition to the above evolution, we mark it as a potential direction to fully automating the learning phase, without the manual feature extraction step. We may utilize the deep learning techniques to automatically extract the features and train an evolutionary classifier to identify the parasitic malware.

It is also notable that, the proposed approach are based on code clustering, without any prior knowledge like the known behavior signatures, but we still require labeled samples to learn suitable parameters, e.g., the threshold and the weights. In the future, we will explore some other features that can better describe the relations among modules such that the approach can throw the parameters away and achieve completely unsupervised.

Besides, we cannot acquire many UEFI rootkits except the Hacking Team's samples. This condition prevents us testing our solution in more different types of rootkits. While we present an acceptable threshold to determine whether a given sample is infected, we might fail the detection if a rootkit is specially designed for a series of uniform machines, in which scenario the rootkit modules do not consist of many conditional checks in contrast to the `ntfs` module, and thus the rootkit modules are not necessarily to be top ranked. And due to the small size of the dataset, we calculated the threshold with only three sample pairs. We would like to explore the applicability of the approach and learn a more robust threshold as done in piggybacked Android application detection when more samples are available.

At last, though we propose a general idea to detect the parasitic malware, based on code clustering and the maximum distance measurement, we have different implementations for different categories of targets. The idea can be applied to other types of parasitic malware, e.g., traditional desktop malware, but we can expect a new implementation to conduct the detection. The implementation should consist of different preprocessing stage, extract target specific features and propose a probably different distance calculation method. However, we think the core of our current implementations can be borrowed to handle the other types of parasitic malware. For example, the features for detecting piggybacked Android applications can be used in other types of applications developed in object-oriented programming

languages, such as Python, Java and C++. The UEFI related feature extraction can also direct future research to quickly discover the target specific features in applications with similar module communication patterns.

5. Related work

Though general malware detection approaches can be employed to detecting parasitic malware [11,36,87]. In this paper, we focus our discussion on two aspects, corresponding to the two detection targets.

5.1. UEFI securities

Before UEFI, BIOS had been the only firmware standard for decades and the attacks against BIOS had never stopped. As demonstrated in [30], most of the attacks were targeting the key components of the boot process, i.e., *Master Boot Record* (MBR) [42,44,45,66], *Volume Boot Record* (VBR) [59,60,62] and the *bootloader* [43]. UEFI adopts a complete different design, which no longer contains MBR, VBR, bootloader or any other BIOS related components. The boot process of UEFI consists of seven phases, namely, the *Security* phase (SEC), the *Pre-EFI Initialization* phase (PEI), the *Driver Execution Environment* phase (DXE), the *Boot Device Selection* phase (BDS), the *Transient System Load* phase (TSL), the *Run time* phase (RT) and last the *After life* phase (AL). Thus, the attacks as well as the defenses [12,30,32] on BIOS can not be applied to UEFI. Nevertheless, curious researchers have designed new attack and defense technologies.

The attacks against the UEFI firmware can be roughly divided into two categories, bootkits and rootkits. A bootkit will hook the normal boot process and try to execute malicious code before the operating system starts up, while a rootkit can be even more troublesome by injecting code directly into the firmware and persisting in the system stealthily. Allievi [1] managed to replace the original `bootmgfw.efi` file (used to load the Windows system) for executing malicious code at the boot time. This bootkit could bypass Windows 8 *Driver Signature Enforcement* and *Patchguard* security mechanism. Kaczmarek [39] proposed a similar hook-based bootkit attack, which hooked a series of boot files, such as the `bootmgfw.efi`, `winload.efi`, `smss.exe` and so on. Wojtczuk and Kallenberg [83,84] leveraged the UEFI boot script to perform a bootkit attack. The boot script is a data structure interpreted by the UEFI firmware. It consists of a sequence of instructions, each of which contains a opcode and the corresponding operands. Wojtczuk and Kallenberg demonstrated that they could run arbitrary code after inserting a custom instruction, or modifying the existing instructions. The UEFI variables are another target for attackers to implement bootkits [6,7,40,41]. The UEFI variables store persistent configurations (e.g., the boot order) of the firmware. The boot process can be interfered by modifying the variables if they are writable. For example, replacing the boot option with customized loaders can result in unauthorized code execution [7]. The *System Management Mode* (SMM) is a special operating mode with high privilege and is designed for use only by the system firmware. It can handle maintenance tasks like power management and system hardware monitoring. Due to the high privilege, it has been a popular target for firmware bootkits. A lot of work leveraged various flaws of SMM to perform attacks [17,18,20,61,64]. For example, Duflot et al. [18] identified four fundamental design flaws of SMM and leveraged the flaws to run arbitrary code in SMM. The aforementioned attacks are all about bootkits, rootkit attacks are far more rare. In 2007, someone with a nickname Icelord first demonstrated the possibility of injecting code into the firmware and implemented a Proof-of-Concept rootkit targeted certain Award BIOS [53]. Several years later, the first BIOS rootkit *Mebromi* was discovered in the wild [26], which is also specifically

targeted at the Award BIOS. The Hacking Team's rootkit is the first practical UEFI rootkit. Recently, the ESET security community exposed a new UEFI rootkit called *Lojax* [21], which is believed to be the trojanized version of *Lojack*, a anti-theft benign UEFI module designed by Absolute Software Corporation. In addition, it is discovered that Lojax leveraged the `ntfs` module of Hacking Team's rootkit.

Faced with so many attacks, security researchers have also come up with some defenses. In fact, the *Secure Boot* mechanism of the UEFI specification [73] can already resist most of the attacks as long as it is correctly implemented. Secure Boot verifies that all running softwares are trusted by the *Original Equipment Manufacturer* (OEM) during the boot process. When the system is powered up, the firmware will check the signature of each boot software, from Option Roms, UEFI modules to the operating system. Only if all the signatures are valid, will the operating system start. However, as demonstrated by Bulygin et al. [8], many manufacturers have implemented the mechanism with flaws, which gives the attackers opportunities to conduct bootkit or rootkit attacks. To mitigate the situation, they proposed 11 rules, such as only allowing signed UEFI firmware updates and correctly implementing signature verification and cryptography functionality, etc. Further, Wilkins and Richardson [82] demonstrated that the system can be more secure when the Secure Boot mechanism is combined with a *TPM* chip on the motherboard, which can securely store confidential information. In 2014, the Intel Security team released a framework test suite called *CHIPSEC* [54]. As they demonstrated, CHIPSEC can be used to analyze the security of computer platforms, such as hardware and firmware.

Despite the defense techniques, we have shown the feasibility of implanting the Hacking Team's rootkit to a valid firmware and executing it successfully. In fact, UEFI rootkit is no longer a theoretical technique with only Proof-of-Concept threats. It can cause serious damages in real world, yet few detection methods have been proposed. In this paper, we devised a lightweight but effective method to detect the UEFI rootkits, as well as to fill this research gap.

5.2. Repackaging detection

In this section, we talk about existing efforts against the repackaging problem, which a broader category of what we focus on in this paper. Android application repackaging is a serious and ongoing security threat to the Android ecosystem. It may directly harm the end-users, violate the developers' rights and contaminate the Android application markets. Thus, detecting repackaged applications is of great importance. By now, plenty of repackaging detection methods have been proposed, which can be broadly divided into three categories, namely, clone detection-based, machine learning-based and runtime monitoring-based methods.

Clone detection-based method. This kind of detection occupies the largest portion of all the existing methods. The basic idea is to find similar applications among a large collection. Zhou et al. [93] implemented an application similarity measurement system called DroidMOSS, which extracted instruction sequences at bytecode level as features. DroidMOSS applied a *fuzzy hashing* technique on each instruction sequence to generate a fingerprint (string) for each application and calculated the edit distance pairwise to find repackaged applications. Glanz et al. [27] proposed a similar system called CodeMatch. Instead of extracting raw instructions, they represented each class by its Android API type list. All the extracted items were organized in alphabetical order. Then, as done in [93], they would fuzzy hash the entire representation of an application and compare the similarities by calculating the edit distance between the according hash values. If the similarity exceeds a predefined threshold, the corresponding applications will be marked as repackaged. In [28], the authors leveraged two kinds of features for comparison, the meta information from the *META-INF* folder and the code information from the .dex file.

The code was characterized by the *n-gram* model. The extracted features are then abstracted in a feature vector for further similarity comparison. Linares-Vásquez et al. [52] also abstracted each application into a feature vector. The features they selected include API calls, identifiers, user permissions, sensors and intents. In addition, they leveraged multiple information retrieval techniques such as *Latent Semantic Indexing* (LSI), *Singular Value Decomposition* (SVD) and *Term-Document-Matrix* (TDM), to process the vectors to get more accurate similarity comparison results. Guan et al. [31] proposed a semantic-based approach called *RepDetector* to detect repackaging applications. RepDetector first identified core methods (methods that are from neither Android libraries nor any third-party libraries), and then obtained the input-output relations of each core method with symbolic execution. The semantic equivalence between two core methods was measured by their output states. At last, they utilized the Mahalanobis distance to quantify the similarity between two applications.

Some research groups valued the code control flow information and devised their detection methods accordingly. Chen et al. [9] constructed the textitControl Flow Graph (CFG) for each method in the app, and then converted each CFG to a self-defined structure called *3D-CFG*. Next, they calculated the centroids of the 3D-CFGs, and leveraged the centroids to measure the similarity between methods. Applications that shared enough similar methods would be recognized as clones. Inspired by [9], Marastoni et al. [58] also leveraged 3D-CFGs and the centroids to find clone applications. To achieve better performance, they considered the invoked APIs as another feature and calculated the application similarities based on the two features. Sun et al. [69] presented a new control-flow representation called *component-based control flow graph* (CB-CFG), of which nodes were Android APIs and edges reflected the control flow precedence relations of these APIs. An application was then represented as a signature which contained a developer identifier and a set of CB-CFGs. They measured the similarity between applications with self-defined metrics.

Lots of researches have demonstrated a fact that repackaged applications always keep the "look" and "feel" similar or identical to the original and they try to utilize such similarities to locate repackaged applications. In [89], the authors addressed that Android applications are user interaction intensive and event dominated. Such observations inspired them to design a new application birthmark, called *feature view graph*. A feature view graph consists of activity classes (nodes) that are associated with potential UI views and the switch relationships (edges) among the activities. Each node and edge is bond with certain features. Repackaged applications can then be detected by measuring the similarity between two feature view graphs using the *VF2* subgraph isomorphism algorithm. Zhauniarovich et al. [91] believed that in order to maintain the "appearance", repackaged applications should have similar resources as the original ones. Thus, they measured the similarity of two apps by calculating the proportion of the number of resource files that exist in both applications to the number of resource files that exist in either application. Namely, the similarity was measured by the *Jaccard* similarity coefficient. Lyu et al. [56], on the other hand, measured the "appearance" similarity using layout files. They implemented a tool called SUIDroid to extract the layout information from all the layout files in an application and organize the information into a compact layout tree. Then, they applied the *AP-TED* (All Path Tree Edit Distance) algorithm to calculate the similarity between two layout trees. The more similar the two trees are, the more likely the corresponding two applications are clones. Sun et al. [68] also leveraged the layout files to compare applications. However, when there were no layout files in an application or the number of elements in layout files was not enough, they would use *perceptual hash values* (pHash) of images in `/res/drawable` folder to represent an app, and further used the hashes to perform similarity comparisons. Similarly, Jiao et al. [37] extracted images from applications and also used the *pHash* algorithm to generate features. By calculating the percentage of identical hash values of two applications, they could

decide whether the applications are clones. Unlike the above work, Chen et al. [10] extracted visual widgets such as Button, ListView, TextView, and the transitions among these widgets, as features. They used such features to construct a view graph for each application and calculated a centroid for each graph to perform pairwise comparisons.

A common limitation of clone detection-based methods is that pairwise comparison will introduce massive computations, especially when the collection is large. To reduce the time cost as well as to improve the scalability of the detection, many methods tend to work in two-phase. The first phase can quickly find out the applications that are not clones so as to reduce the number of applications and the second phase will identify true clones. For example, Crussell et al. [14] presented a detection system called DNADroid, which first selected potential application clones based on several application attributes, e.g., the application name, package, market, owner and descriptions. Then, it constructed *Program Dependency Graph* (PDG) for each selected application and leveraged the VF2 algorithm to compute the similarity between two PDGs. Zhou et al. [92] aimed at detecting piggybacked applications. In the first stage, they proposed *module decoupling* technique to partition an application's code into primary and non-primary modules. The primary modules contain code that achieves the core functionality of the application, while the non-primary modules refer to the third-party libraries or piggybacked code. It is believed that the piggybacked application should contain the same primary modules as the original one does. Then in the second stage, they extracted various features such as requested permissions, used Android API calls and involved intent type from the primary modules. These features are then converted into vectors for similarity comparisons. Wang et al. [77] also presented a two-phase approach called Wukong. In the first coarse-grained phase, Wukong simply extracted the call frequencies of different Android APIs to select potential clones, and in the next fine-grained phase, it counted the number of times each variable occurs in different contexts and leveraged these numbers to identify clones from the candidates. To achieve better detection scalability, Lyu et al. proposed *FUIDroid* [55], an enhanced tool of SUIDroid [56]. FUIDroid first leveraged some natural language processing techniques to compare applications' descriptions to select potential cloned applications and then utilized SUIDroid to verify which potential clones are true.

Maintaining an application repository large enough is non-trivial. Our approach does not require the original counterparts of piggybacked applications for detection. Instead, it leverages only the information within the target application.

Machine learning-based method. Recent years, machine learning has been in the spotlight, because of its excellent performance in many tasks, such as image classification, natural language processing and speech recognition. Naturally, some researchers explored the possibility of using machine learning techniques to detect repackaged applications. Shao et al. [65] proposed a two-stage methodology. In the first coarse-grained stage, they extracted 15 statistical features, e.g., number of activities and number of intent filters, to represent each application as a 15-dimensional vector, and calculated the Euclidean distance between two vectors to locate repackaged candidates. In the second stage, they extracted two types of structural features, which were the activity layouts and the event handlers. They mapped the two features into vectors and applied either *spectral clustering* algorithm or *nearest neighbor search* (NNS) to find similar apps among the candidates. To avoid comparing applications pairwise, Crussell et al. [15] proposed a scalable approach called Andarwin. Andarwin first constructed PDGs for methods and then extracted semantic information from the PDGs. All the semantic information would be abstracted into feature vectors. At last, they applied the *Locality Sensitive Hashing* (LSH) algorithm to find nearest neighbors among all the vectors and the similar applications can be obtained accordingly. In [22], Fan et al. leveraged the distinguished invocation patterns of sensitive APIs between host code and injected

code to detect piggybacked applications. In particular, they constructed a *sensitive subgraph* (SSG) from an application's *call graph*. The SSG could profile the most suspicious behavior of an application. Then, they extracted five self-defined features from the SSG to characterize the application. At last, they trained four machine learning classifiers (*Random Forest*, *Decision Tree*, *k-NN* and *PART*) with the five features and used the trained models to predict whether an input application is piggybacked. Instead of training with features of an entire application, Tian et al. [70] partitioned the code into different regions according to some class-level relations and method-level relations, and further extracted three types of features (user interaction features, sensitive API features and permission request features) from each region. Then, they trained classifiers for regions. If an application contained malicious regions, it was considered repackaged.

Similar to the clone detection based approaches, machine learning based methods require a large repository for training while our approach does not depend on the applications other than the target itself.

Runtime monitoring-based method. Both aforementioned methods extract static features for detection. Some researches have focused on devising detection methods using dynamic (runtime) features. Lin et al. [51] proposed a detection mechanism called *SCSdroid* (System Call Sequence droid). As the name implies, SCSdroid characterized each application with its thread-grained system call sequence during runtime. For detection, SCSdroid adopted the *Bayes Theorem* to calculate the possibility of a call sequence being malicious and then further decided whether the application is repackaged. Soh et al. [67] proposed a detection based on the analysis of runtime user interface information. Specifically, given an application, they extracted the names of activities and executed the non-library classes to collect the UI information, which was stored in XML format. Then, they generated a birthmark from the XML file and using the LSH algorithm to find neighbors among a collection of applications. Yue et al. [88] also leveraged the runtime UI information to characterize applications. They proposed a structure called *layout-group graph* (LGG) as dynamic birthmark of an application, which was constructed from the runtime UI traces and applications who shared similar LGGs are considered repackaged. Wu et al. [85] extracted features from the HTTP traffic that generated by a running application. They parsed the HTTP traffic into HTTP flows using a six-tuple (host, method, page, parameter, value, type) and filtered out the library traffic. Then, they used the *HTTP Flow Distance* algorithm and *Hungarian Method* algorithm to calculate similarity between two HTTP flows and repackaged applications were identified if the similarity score was high.

Dynamically collecting the signatures has its own limitation of efficiency and completeness. In addition, some techniques also demand many other applications for comparison while our approach performs detection constricted in each single application.

6. Conclusion

In this paper, we propose a code clustering-based malware detection mechanism which requires no prior knowledge of the target malware. Based on the mechanism, we devise a two-stage detection approach for two representative kinds of malware, the UEFI rootkits and the piggybacked Android applications. For both types of malware, the approach first clusters the modules according to the interactive relations extracted from the code. The outliers are exposed. Then distances between the outliers and the largest cluster are computed in UEFI rootkit detection. When a given UEFI firmware shows a larger distance than an empirical threshold, it is reported as infected. The second stage in piggybacked Android

application detection differs. It leverages the organizational similarities of the modules to aggregate the outliers into larger clusters. If at last multiple clusters exist, i.e., the maximum outlier distance is non-zero, piggybacking is reported. We implement the prototypes and evaluate them on existing real-world malware samples. The results show a good performance of our approach. All infected UEFI firmwares are detected without false negatives and all benign samples are identified without false warnings. The approach also achieves a high *F1* score (90.66%) for piggybacked Android application detection when the popular third-party libraries are eliminated, with both the precision rate and recall rate above or near 90%. While the experiments illustrate an acceptable performance, the approach has some deficiencies as discussed in Section 4 and we leave them in future work to explore more valuable features and advanced techniques as well as to apply the approach on more types of malware.

Acknowledgments

This work is supported in part by National Natural Science Foundation of China (NSFC) under grants U1836209 and 61802413, the Fundamental Research Funds for the Central Universities, and the Research Funds of Renmin University of China under grant 19XNLG02. Any opinions, findings, and conclusions in this paper are those of the authors only and do not necessarily reflect the views of our sponsors.

References

[1] A. Allievi, UEFI technology: Say hello to the Windows 8 bootkit, 2012.
[2] K. Allix, T.F. Bissyandé, J. Klein and Y. Le Traon, AndroZoo: Collecting millions of Android apps for the research community, in: *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)*, IEEE, 2016, pp. 468–471.
[3] K. Arnold, J. Gosling and D. Holmes, *The Java Programming Language*, Addison Wesley Professional, 2005, pp. 43–62.
[4] Asus, Asus support, https://asus-drivers-download-center.blogspot.com/2018/11/asus-x550c-drivers-for-windows-8-64-bit.html, accessed 16 March 2019.
[5] M. Backes, S. Bugiel and E. Derr, Reliable third-party library detection in Android and its security applications, in: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ACM, 2016, pp. 356–367.
[6] O. Bazhaniuk, Y. Bulygin, A. Furtak, M. Gorobets, J. Loucaides, A. Matrosov and M. Shkatov, Attacking and defending BIOS in 2015, in: *ReCon Conference*, 2015, http://www.intelsecurity.com/advanced-threat-research/content/AttackingAndDefendingBIOS-RECon2015.pdf.
[7] A. Boursalian, Bootbandit: A macOS bootloader attack, 2017.
[8] Y. Bulygin, A. Furtak and O. Bazhaniuk, A tale of one software bypass of Windows 8 Secure Boot, Black Hat USA, 2013.
[9] K. Chen, P. Liu and Y. Zhang, Achieving accuracy and scalability simultaneously in detecting application clones on Android markets, in: *Proceedings of the 36th International Conference on Software Engineering*, ACM, 2014, pp. 175–186.
[10] K. Chen, P. Wang, Y. Lee, X. Wang, N. Zhang, H. Huang, W. Zou and P. Liu, Finding unknown malice in 10 seconds: Mass vetting for new threats at the Google-play scale, in: *24th {USENIX} Security Symposium ({USENIX} Security 15)*, 2015, pp. 659–674.
[11] M. Christodorescu, S. Jha, S.A. Seshia, D. Song and R.E. Bryant, Semantics-aware malware detection, in: *2005 IEEE Symposium on Security and Privacy*, IEEE, 2005, pp. 32–46. doi:10.1109/SP.2005.20.
[12] D. Cooper, W. Polk, A. Regenscheid and M. Souppaya, BIOS protection guidelines, NIST Special Publication, 2011.
[13] I.D. Corporation, Smartphone market share, https://www.idc.com/promo/smartphone-market-share/os, accessed 22 March 2019.
[14] J. Crussell, C. Gibler and H. Chen, Attack of the clones: Detecting cloned applications on Android markets, in: *European Symposium on Research in Computer Security*, Springer, 2012, pp. 37–54.
[15] J. Crussell, C. Gibler and H. Chen, AnDarwin: Scalable detection of Android application clones based on semantics, *IEEE Transactions on Mobile Computing* **14**(10) (2015), 2007–2019. doi:10.1109/TMC.2014.2381212.

[16] Dell, Dell support, https://www.dell.com/support/home/bg/en/bgbsdt1/product-support/product/latitude-e6320/drivers, https://www.dell.com/support/home/bg/en/bgbsdt1/product-support/product/precision-t1600/drivers, accessed 16 March 2019.
[17] C. Domas, The memory sinkhole – Unleashing an x86 design flaw allowing universal privilege escalation, Black Hat USA, 2015.
[18] L. Duflot, O. Levillain, B. Morin and O. Grumelard, System management mode design and security issues, IT Defense, 2010.
[19] C. Eagle, *The IDA Pro Book*, 2nd edn, No Starch Press, 2011.
[20] S. Embleton, S. Sparks and C.C. Zou, SMM rootkit: A new breed of OS independent malware, *Security and Communication Networks* **6**(12) (2013), 1590–1605. doi:10.1002/sec.166.
[21] ESET Research, LOJAX: First UEFI rootkit found in the wild, courtesy of the Sednit group, 2018, https://www.welivesecurity.com/wp-content/uploads/2018/09/ESET-LoJax.pdf.
[22] M. Fan, J. Liu, W. Wang, H. Li, Z. Tian and T. Liu, DAPASA: Detecting Android piggybacked apps through sensitive subgraph analysis, *IEEE Transactions on Information Forensics and Security* **12**(8) (2017), 1772–1785. doi:10.1109/TIFS.2017.2687880.
[23] Q. Feng, R. Zhou, C. Xu, Y. Cheng, B. Testa and H. Yin, Scalable graph-based bug search for firmware images, in: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ACM, 2016, pp. 480–491.
[24] Github, EFI scripts for IDA Pro, https://github.com/snare/ida-efiutils, accessed 16 March 2019.
[25] Github, UEFITool, https://github.com/LongSoft/UEFITool, accessed 16 March 2019.
[26] M. Giuliani, Mebromi: The first BIOS rootkit in the wild, 2011, http://blog.webroot.com/2011/09/13/mebromi-thefirst-bios-rootkit-in-the-wild/.
[27] L. Glanz, S. Amann, M. Eichberg, M. Reif, B. Hermann, J. Lerch and M. Mezini, CodeMatch: Obfuscation won't conceal your repackaged app, in: *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ACM, 2017, pp. 638–648.
[28] H. Gonzalez, N. Stakhanova and A.A. Ghorbani, Droidkin: Lightweight detection of Android apps similarity, in: *International Conference on Security and Privacy in Communication Systems*, Springer, 2014, pp. 436–453.
[29] Google Developers, The online Android developers documentation, https://developer.android.com/docs, accessed 14 April 2019.
[30] B. Grill, A. Bacs, C. Platzer and H. Bos, "Nice Boots!" – A large-scale analysis of bootkits and new ways to stop them, in: *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, Springer, 2015, pp. 25–45. doi:10.1007/978-3-319-20550-2_2.
[31] Q. Guan, H. Huang, W. Luo and S. Zhu, Semantics-based repackaging detection for mobile apps, in: *International Symposium on Engineering Secure Software and Systems*, Springer, 2016, pp. 89–105. doi:10.1007/978-3-319-30806-7_6.
[32] L. Haukli, Exposing bootkits with BIOS emulation, Black Hat USA, 2014.
[33] S. Heron, Parasitic malware: The resurgence of an old threat, *Network Security* **2008**(3) (2008), 15–18. doi:10.1016/S1353-4858(08)70032-1.
[34] J. Huang, X. Zhang, L. Tan, P. Wang and B. Liang, AsDroid: Detecting stealthy behaviors in Android applications by user interface and program behavior contradiction, in: *Proceedings of the 36th International Conference on Software Engineering*, ACM, 2014, pp. 1036–1046.
[35] Intel Corporation, *EDK II Module Writer's Guide, Version 0.7*, 2010, pp. 50–58, https://github.com/tianocore/tianocore.github.io/wiki/EDK-II-User-Documentation.
[36] X. Jiang, X. Wang and D. Xu, Stealthy malware detection through VMM-based out-of-the-box semantic view reconstruction, in: *Proceedings of the 14th ACM Conference on Computer and Communications Security*, ACM, 2007, pp. 128–138.
[37] S. Jiao, Y. Cheng, L. Ying, P. Su and D. Feng, A rapid and scalable method for Android application repackaging detection, in: *Information Security Practice and Experience*, Springer, 2015, pp. 349–364. doi:10.1007/978-3-319-17533-1_24.
[38] J.-H. Jung, J.Y. Kim, H.-C. Lee and J.H. Yi, Repackaging attack on Android banking applications and its countermeasures, *Wireless Personal Communications* **73**(4) (2013), 1421–1437. doi:10.1007/s11277-013-1258-x.
[39] S. Kaczmarek, UEFI and Dreamboot, in: *Hack in the Box Security Conference*, Kuala Lumpur, Malasia, 2013.
[40] C. Kallenberg, S. Cornwell, X. Kovah and J. Butterworth, Setup for failure: Defeating Secure Boot, in: *The Symposium on Security for Asia Network (SySCan)*, 2014.
[41] C. Kallenberg, X. Kovah, J. Butterworth and S. Cornwell, Extreme privilege escalation on Windows 8/UEFI systems, Black Hat USA, 2014.
[42] P. Kleissner, Stoned bootkit, Black Hat USA, 2009.
[43] Krebs on Security, Carberp code leak stokes copycat fears, https://krebsonsecurity.com/tag/carberp-source-code-leak/, accessed 19 March 2019.
[44] N. Kumar and V. Kumar, Vbootkit: Compromising Windows Vista security, Black Hat Europe, 2007.

[45] N. Kumar and V. Kumar, VBootKit 2.0 – Attacking Windows 7 via Boot Sectors, in: *Proceedings of the Hack in the Box Conference (HITBSecConf)*, 2009.
[46] L. Li, T.F. Bissyande and J. Klein, Rebooting research on detecting repackaged Android apps: Literature review and benchmark, *IEEE Transactions on Software Engineering* (2019). doi:10.1109/TSE.2019.2901679.
[47] L. Li, T.F. Bissyandé, J. Klein and Y. Le Traon, An investigation into the use of common libraries in Android apps, in: *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, Vol. 1, IEEE, 2016, pp. 403–414.
[48] L. Li, D. Li, T.F.D.A. Bissyande, D. Lo, J. Klein and Y. Le Traon, Ungrafting malicious code from piggybacked Android apps, Technical report, SnT, 2016.
[49] M. Li, W. Wang, P. Wang, S. Wang, D. Wu, J. Liu, R. Xue and W. Huo, LibD: Scalable and precise third-party library detection in Android markets, in: *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, IEEE, 2017, pp. 335–346. doi:10.1109/ICSE.2017.38.
[50] P. Lin, Hacking Team uses UEFI BIOS rootkit to keep RCS 9 agent in target systems, Trend Micro, 2015, https://blog.trendmicro.com/trendlabs-security-intelligence/hacking-team-uses-uefi-bios-rootkit-to-keep-rcs-9-agent-in-target-systems/.
[51] Y.-D. Lin, Y.-C. Lai, C.-H. Chen and H.-C. Tsai, Identifying Android malicious repackaged applications by thread-grained system call sequences, *Computers & Security* **39** (2013), 340–350. doi:10.1016/j.cose.2013.08.010.
[52] M. Linares-Vásquez, A. Holtzhauer and D. Poshyvanyk, On automatically detecting similar Android apps, in: *2016 IEEE 24th International Conference on Program Comprehension (ICPC)*, IEEE, 2016, pp. 1–10.
[53] I. Lord, BIOS rootkit: Welcome home, my Lord!, 2007, https://blog.csdn.net/icelord/article/details/1604884.
[54] J. Loucaides and Y. Bulygin, Platform security assessment with CHIPSEC, in: *CanSecWest Applied Security Conference (CanSecWest 2014)*, 2014.
[55] F. Lyu, Y. Lin and J. Yang, An efficient and packing-resilient two-phase Android cloned application detection approach, *Mobile Information Systems* **2017** (2017), 6958698. doi:10.1155/2017/6958698.
[56] F. Lyu, Y. Lin, J. Yang and J. Zhou, Suidroid: An efficient hardening-resilient approach to Android app clone detection, in: *2016 IEEE Trustcom/BigDataSE/ISPA*, IEEE, 2016, pp. 511–518. doi:10.1109/TrustCom.2016.0104.
[57] Z. Ma, H. Wang, Y. Guo and X. Chen, LibRadar: Fast and accurate detection of third-party libraries in Android apps, in: *Proceedings of the 38th International Conference on Software Engineering Companion*, ACM, 2016, pp. 653–656.
[58] N. Marastoni, A. Continella, D. Quarta, S. Zanero and M.D. Preda, GroupDroid: Automatically grouping mobile malware by extracting code similarities, in: *Proceedings of the 7th Software Security, Protection, and Reverse Engineering/Software Security and Protection Workshop*, ACM, 2017, p. 1.
[59] A. Matrosov, Olmasco bootkit: Next circle of TDL4 evolution (or not?), ESET, 2012.
[60] A. Matrosov, Rovnix bootkit framework updated, 2012.
[61] J. Rauchberger, R. Luh and S. Schrittwieser, Longkit – A universal framework for BIOS/UEFI rootkits in system management mode, in: *ICISSP*, 2017, pp. 346–353.
[62] E. Rodionov, Win32/Gapz: New bootkit technique, 2012.
[63] M. Rothman, V. Zimmer and T. Lewis, *Harnessing the UEFI Shell: Moving the Platform Beyond DOS*, Walter de Gruyter GmbH & Co KG, 2017. doi:10.1515/9781501505751.
[64] J. Schiffman and D. Kaplan, The SMM rootkit revisited: Fun with USB, in: *2014 Ninth International Conference on Availability, Reliability and Security*, IEEE, 2014, pp. 279–286. doi:10.1109/ARES.2014.44.
[65] Y. Shao, X. Luo, C. Qian, P. Zhu and L. Zhang, Towards a scalable resource-driven approach for detecting repackaged Android applications, in: *Proceedings of the 30th Annual Computer Security Applications Conference*, ACM, 2014, pp. 56–65.
[66] D. Soeder and R. Permeh, eEye BootRoot, Black Hat USA, 2005.
[67] C. Soh, H.B.K. Tan, Y.L. Arnatovich and L. Wang, Detecting clones in Android applications through analyzing user interfaces, in: *Proceedings of the 2015 IEEE 23rd International Conference on Program Comprehension*, IEEE Press, 2015, pp. 163–173. doi:10.1109/ICPC.2015.25.
[68] M. Sun, M. Li and J. Lui, DroidEagle: Seamless detection of visually similar Android apps, in: *Proceedings of the 8th ACM Conference on Security & Privacy in Wireless and Mobile Networks*, ACM, 2015, p. 9.
[69] X. Sun, Y. Zhongyang, Z. Xin, B. Mao and L. Xie, Detecting code reuse in Android applications using component-based control flow graph, in: *IFIP International Information Security Conference*, Springer, 2014, pp. 142–155.
[70] K. Tian, D.D. Yao, B.G. Ryder, G. Tan and G. Peng, Detection of repackaged Android malware with code-heterogeneity features, *IEEE Transactions on Dependable and Secure Computing* **17**(1) (2017), 64–77. doi:10.1109/TDSC.2017.2745575.
[71] TianoCore, EDK II project, 2019, https://github.com/tianocore/edk2.
[72] C. Tumbleson and R. Wiśniewski, A tool for reverse engineering Android apk files, 2019, https://ibotpeaches.github.io/Apktool/, accessed 5 March 2019.

[73] Unified EFI Forum, Inc., *Unified Extensible Firmware Interface Specification, Version 2.7*, 2017, https://uefi.org/sites/default/files/resources/UEFI_Spec_2_7.pdf.
[74] Unified Extensible Firmware Interface Forum, Inc., *Platform Initialization (PI) Specification, Version 1.7*, 2019, pp. 526–540, https://uefi.org/sites/default/files/resources/PI_Spec_1_7_final_Jan_2019.pdf.
[75] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam and V. Sundaresan, Soot: A Java bytecode optimization framework, in: *CASCON First Decade High Impact Papers*, IBM Corp., 2010, pp. 214–224. doi:10.1145/1925805.1925818.
[76] VirusTotal, Analyze suspicious files and URLs to detect types of malware, automatically share them with the security community, https://www.virustotal.com/#/home/upload, accessed 14 March 2019.
[77] H. Wang, Y. Guo, Z. Ma and X. Chen, Wukong: A scalable and accurate two-phase approach to Android app clone detection, in: *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ACM, 2015, pp. 71–82. doi:10.1145/2771783.2771795.
[78] F. Wei, Y. Li, S. Roy, X. Ou and W. Zhou, Deep ground truth analysis of current Android malware, in: *DIMVA*, 2017.
[79] WikiLeaks, Infected machines, https://wikileaks.org/hackingteam/emails/emailid/19404, accessed 7 March 2019.
[80] WikiLeaks, UEFI Windows persistent, https://wikileaks.org/hackingteam/emails/emailid/526357, accessed 7 March 2019.
[81] Wikipedia, Hacking Team: 2015 data breach, https://en.wikipedia.org/wiki/Hacking_Team#2015_data_breach, accessed 7 March 2019.
[82] R. Wilkins and B. Richardson, UEFI secure boot in modern computer security solutions, in: *UEFI Forum*, 2013.
[83] R. Wojtczuk and C. Kallenberg, Attacking UEFI boot script, in: *31st Chaos Communication Congress (31C3)*, 2014.
[84] R. Wojtczuk and C. Kallenberg, Attacks on UEFI security, in: *Proc. 15th Annu. CanSecWest Conf. (CanSecWest)*, 2015.
[85] X. Wu, D. Zhang, X. Su and W. Li, Detect repackaged Android application based on http traffic similarity, *Security and Communication Networks* **8**(13) (2015), 2257–2266. doi:10.1002/sec.1170.
[86] C. Xiao, Novel malware XcodeGhost modifies Xcode, infects apple iOS apps and hits app store, Technical report, 2015, https://unit42.paloaltonetworks.com/novel-malware-xcodeghost-modifies-xcode-infects-apple-ios-apps-and-hits-app-store/.
[87] H. Yin, D. Song, M. Egele, C. Kruegel and E. Kirda, Panorama: Capturing system-wide information flow for malware detection and analysis, in: *Proceedings of the 14th ACM Conference on Computer and Communications Security*, ACM, 2007, pp. 116–127.
[88] S. Yue, W. Feng, J. Ma, Y. Jiang, X. Tao, C. Xu and J. Lu, RepDroid: An automated tool for Android application repackaging detection, in: *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*, IEEE, 2017, pp. 132–142. doi:10.1109/ICPC.2017.16.
[89] F. Zhang, H. Huang, S. Zhu, D. Wu and P. Liu, ViewDroid: Towards obfuscation-resilient mobile application repackaging detection, in: *Proceedings of the 2014 ACM Conference on Security and Privacy in Wireless & Mobile Networks*, ACM, 2014, pp. 25–36.
[90] Y. Zhang, R. Jin and Z.-H. Zhou, Understanding bag-of-words model: A statistical framework, *International Journal of Machine Learning and Cybernetics* **1**(1–4) (2010), 43–52. doi:10.1007/s13042-010-0001-0.
[91] Y. Zhauniarovich, O. Gadyatskaya, B. Crispo, F. La Spina and E. Moser, FSquaDRA: Fast detection of repackaged applications, in: *IFIP Annual Conference on Data and Applications Security and Privacy*, Springer, 2014, pp. 130–145.
[92] W. Zhou, Y. Zhou, M. Grace, X. Jiang and S. Zou, Fast, scalable detection of piggybacked mobile applications, in: *Proceedings of the Third ACM Conference on Data and Application Security and Privacy*, ACM, 2013, pp. 185–196.
[93] W. Zhou, Y. Zhou, X. Jiang and P. Ning, Detecting repackaged smartphone applications in third-party Android marketplaces, in: *Proceedings of the Second ACM Conference on Data and Application Security and Privacy*, ACM, 2012, pp. 317–326.