Combating the OS-level Malware in Mobile Devices by Leveraging Isolation and Steganography

Niusen Chen, Wen Xie, and Bo Chen

Department of Computer Science
Michigan Technological University
{niusenc, wenxie, bchen}@mtu.edu

Abstract. Detecting the OS-level malware (e.g., rootkit) is an especially challenging problem, as this type of malware can compromise the OS, and can then easily hide their intrusion behaviors or directly subvert the traditional malware detectors running in either the user or the kernel space. In this work, we propose mobiDOM to solve this problem for mobile computing devices. The key idea of mobiDOM is to securely detect the OS-level malware by fully utilizing the existing secure features of a mobile device in the hardware. Specifically, we integrate a malware detector in the flash translation layer (FTL), a firmware layer embedded into the external flash storage which is inaccessible to the OS; in addition, we build a trusted application in the Arm TrustZone secure world, which acts as a user-level controller of the malware detector. The FTL-based malware detector and the TrustZone-based controller communicate with each other stealthily via steganography. Security analysis and experimental evaluation confirm that mobiDOM can securely and effectively detect the OS-level malware.

Keywords: OS-level malware · Detection · Hardware isolation · Flash translation layer · TrustZone · Steganography

1 Introduction

We have been witnessing a surge of malware for mobile devices in the past few years [23]. The malware intrudes into a victim mobile device, stealing personally private or even mission critical data, corrupting the local storage [24], or controlling the entire victim device. Especially, there is one type of strong malware (e.g., rootkit) which is able to compromise the entire operating system of the device, obtaining the root privilege. This type of OS-level malware is extremely difficult to be combated, since it can easily subvert any traditional anti-malware software or tools running in the user/ kernel space [6, 12, 14, 15, 21, 30], by leveraging its high privilege.

To combat the OS-level malware, a first step is to detect them once they are present in the victim mobile devices. This requires a malware detector, which

can monitor the system and, once any abnormal activities happen, the malware detector will make a decision and inform the user (e.g., via a user app) if it reaches a "malware detected" decision. It turns out that the malware detector cannot simply run in the normal execution environment of the device to avoid being compromised by the OS-level malware. In addition, the user app which interacts with both the end users and the malware detector should not be compromised by the OS-level malware either. Therefore, a key idea towards a secure design is to place both the malware detector and the user app to an execution environment which is isolated from the regular OS and hence the OS-level malware.

Compared to traditional desktops/ laptops, mobile computing devices today are equipped with unique hardware features: 1) They usually use ARM processors which have reduced circuit complexity and low power consumption and, ARM processors have integrated **TrustZone**, a hardware security feature, into any Cortex-A processor (built on the Armv7-A and Armv8-A architecture) and Cortex-M processors (built on the Armv8-M architecture). TrustZone enables the establishment of a trusted execution environment that is hardware separated from the normal insecure execution environment. 2) They typically use flash storage media instead of hard disk drives (HDD) for external storage. For example, smartphones, tablets, IoT devices extensively use microSD, eMMC, or UFS cards. Different from HDDs, flash memory exhibits different physical nature and traditional file systems built for HDDs cannot directly be used on them. To bridge this gap, a new flash translation layer (**FTL**) is usually incorporated into the flash storage media to transparently handle the unique nature of flash, exposing an HDD-like interface externally.

Leveraging the unique hardware features of mobile devices, we have designed mobiDOM, the first scheme which can securely and effectively Detect the OS-level Malware in the mobile devices. Our insights are three-fold: First, we integrate the malware detector into the FTL. This new design is advantageous because: 1) the OS-level malware will not be able to subvert the malware detector located in the FTL which is isolated by the flash storage hardware and remains transparent to the OS (*security*); and 2) the malware usually needs to perform I/Os on the external storage, and such I/Os may exhibit some unique behaviors which can be observed in the FTL as confirmed in our experiments using real-world malware samples (*effectiveness*). Second, we introduce a user-level controller which can allow the end user to control the malware detector located in the FTL. To prevent the controller from being subverted by the OS-level malware, we separate its functionality and move the critical component into the "trusted execution environment" (i.e., a secure world) established by TrustZone. In this way, the controller is secure and is able to work with the malware detector correctly. Third, to prevent the malware from noticing the communication between the malware detector (staying in the FTL) and the controller (the key component is staying in the secure world), we leverage steganography, so that the controller and the malware detector can communicate stealthily via the regular I/Os performed on the external storage.

Contributions. Major contributions of this work are:

– We have proposed the first framework (mobiDOM) which can securely detect the OS-level malware, utilizing both the isolation environments provided by main-stream mobile computing devices as well as the steganography technique.
– We have developed a prototype of mobiDOM, and ported it to a real-world testbed to assess its performance. We have also assessed the effectiveness of our FTL-based malware detector by collecting real-world malware samples, running it in our testbed to capture the I/O traces in the FTL, and using the I/O traces for training and testing the malware detector.
– We also analyze the security of mobiDOM.

2 Background

2.1 Flash Memory

Flash memory especially NAND flash has been extensively used as the mass storage of main-stream mobile computing devices, in form of SD/ miniSD/ microSD cards, MMC cards, and UFS cards. Compared to traditional mechanical disk drives (e.g., hard disk drives), a flash storage medium removes all the mechanical components, and is electrically erasable and re-programmable. Therefore, it usually has much higher I/O throughput with much lower noise. The NAND flash is usually divided into blocks, with typical block sizes 16KB, 128KB, 256KB, or 512KB. Each block consists of pages, each of which can be 512B, 2KB, or 4KB in size. In general, the read/ write operations in NAND flash are performed on the basis of pages, while the erase operations are performed on the basis of blocks.

NAND flash exhibits a few unique characteristics: First, it follows an erase-then-write design. In other words, a flash block needs to be erased first before it can be re-programmed. Therefore, to modify the data stored in a page, we need to first erase the encompassing block, which requires copying out valid data in this block, erasing the block, and writing the data back, leading to significant write amplification. To mitigate the write amplification, flash memory usually uses an out-of-place instead of in-place update strategy. Second, each flash block only allows a limited number of program-erase (P/E) cycles and, if a flash block is programmed/ erased too frequently, it will turn "bad" and cannot store data correctly. Therefore, wear leveling is needed to distribute P/Es evenly across the entire flash. Third, reading/ writing a flash memory cell frequently may cause its nearby cells in the same block to change over time, causing read/ write disturb errors.

Flash translation layer (FTL). Existing flash storage media usually can be used as block devices just like HDDs. This is because, they usually integrate a flash translation layer (FTL), to transparently handle the special nature of NAND flash, exposing a block-access interface. In this way, traditional block-based file systems (e.g., EXT, FAT, and NTFS) can be directly deployed on top of flash storage media. The FTL usually implements four key functions: address translation, garbage collection, wear leveling and bad block management. Address translation manages the mappings between the block addresses

and the actual flash memory addresses. Garbage collection periodically reclaims the flash blocks which store invalid data (the data are invalidated due to the out-of-place update). Wear leveling ensures that programmings/ erasures are distributed evenly across the flash. Bad block management handles those blocks which occasionally turn "bad", so that they will not be used to store valid data.

2.2 ARM TrustZone

The TrustZone security extensions are available in ARM Cortex-A processors (or processors built on the Armv7-A and Armv8-A architecture), as well as ARM Cortex-M processors (built on the Armv8-M architecture). Its core idea is to create two execution environments which run simultaneously on a single processor: a secure execution environment (i.e., the *secure world*) which can be used to run sensitive applications, and a non-secure execution environment (i.e., the *normal world*) which can be used to run non-sensitive applications. Each world operates independently when using the same processor and, switching between them is orthogonal to all other capabilities of the processor. Memory/ peripherals are aware of the corresponding world of the core and, applications running in the normal world cannot have access to the memory space of the secure world.

2.3 Steganography

Steganography in communication allows a sender to send a seemingly innocuous message, which conceals some critical information, to a receiver. In this way, the critical message can be delivered to the receiver stealthily, i.e., without being noticed by the adversary staying in the middle. Compared to cryptography which protects the content of critical messages (e.g., via encryption), steganography is more advantageous since it essentially conceals the fact that a critical message is being sent as well as protects the corresponding content.

3 System and Adversarial Model

3.1 System Model

We consider a mobile computing device which is equipped with an ARM processor (with TrustZone feature enabled) and a flash-based block device as external storage (e.g., a miniSD/ microSD card, eMMC card, or UFS card). This type of mobile devices can be found widely in real world, including smartphones, tablets, wearable devices, etc. A general architecture of the device is shown in Figure 1. By leveraging TrustZone, we can create a secure world in the mobile device, in which there is a small trusted operating system, with trusted applications (**TA**) running on top of it. In the normal world of the mobile device, there is a rich operating system, with regular applications running on top of it. The external flash storage is used as a block device since the FTL transparently handles the

unique nature of NAND flash, exposing a block access interface externally. We assume there are N data blocks on the block layer which are usable by the OS. Note that the size of a data block is equal to the size of a flash page, and if the flash page is 2KB, each data block will consist of 4 512-byte sectors.

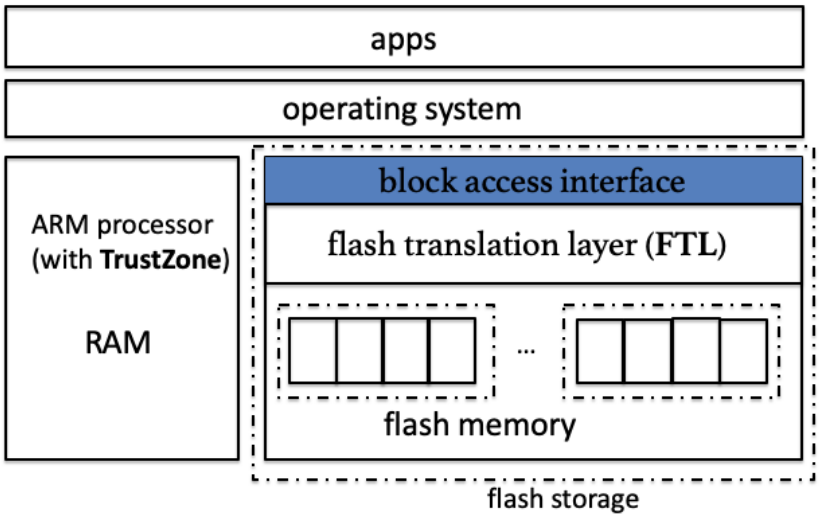

Fig. 1. The architecture of a mobile device

3.2 Adversarial Model

We consider the OS-level malware which is able to compromise the regular rich operating system (in the normal world) of a victim mobile computing device, e.g., by exploiting the system vulnerabilities and escalating the privilege. By compromising the OS, the malware is able to subvert any malware detection tools which run in either the application level or the system level of the normal world. Note that the malware in real world is highly heterogeneous in behaviors, and we only target the malware (e.g., computer viruses, ransomware, etc.) which will perform abnormal/ suspicious I/Os on the external storage.

We rely on a few assumptions: First, the malware is assumed to be not able to compromise the trusted OS and TAs running in the TrustZone secure world, which is a reasonable assumption in the domain of TrustZone technologies [13]. The malware is also assumed to be not able to hack into the FTL, which only provides a limited set of block access interfaces externally to the OS. Second, we assume that the malware will not perform DoS attacks, e.g., blocking regular (or seemingly regular) communications between TAs running in the secure world and applications running in the normal world, or blocking I/Os performed by TAs on the external flash storage device. This assumption is reasonable because: if the malware blocks the communications or I/Os, the TAs in the secure world

will detect such an anomaly trivially and notify the user. However, the malware may view, modify or replay the communicated messages and remain undetected.

4 Design

4.1 Design Overview

The design overview of mobiDOM is shown in Figure 2. We introduce TApp, a trusted application running in the TrustZone secure world. We also introduce MDetector, a malware detector running in the FTL. The TApp acts as a trusted controller of the MDetector. Both the TApp and the MDetector are running in an individual isolated execution environment which is invisible to the operating system (and the OS-level malware as well). A key issue is how to allow the TApp and the MDetector to securely communicate with each other without being compromised by the OS-level malware. An immediate solution is to allow the TApp to read/ write the external flash storage directly via the trusted OS running in the TrustZone. This however, requires adding extra software components (i.e., disk drivers and other necessary components in the storage path) to the small trusted OS which would introduce a lot of extra burden to the secure world.

Having observed that a trusted application in the secure world is usually invoked by a client application (CA) in the normal world, we therefore let the TApp use the CA as a proxy[1] to communicate with the MDetector, taking advantage of the rich OS in the normal world. To protect the communication between the TApp and the MDetector, we leverage steganography. Specifically, the secret messages being communicated between the TApp and the MDetector are hidden in the (seemingly) normal I/Os issued by the TApp on the flash storage device (via CA as a proxy). This is advantageous in a few aspects: First, since the secret communicated messages between the TApp and the MDetector are hidden in the normal I/Os, and their confidentiality can be ensured. Second, the integrity of the secret communicated messages can be also ensured since if the malware manipulates the I/Os, the secret messages will not be extracted correctly by the receiver and the receiver will notice that. Third, the steganography technique essentially hides the fact that some secret messages are exchanged, and therefore, the malware is not able to notice the existence of this cover communication.

A few challenges need to be addressed. First, how can we hide the secret messages (e.g., start, stop, query command issued by the TApp to the MDetector) into the regular I/Os? Our idea is, when the TApp performs a write request on the external storage, we randomly select a portion of bits from the write request to embed a secret message. Note that: 1) Each write request is usually a few KBs in size, and the secret message is 100+ bits[2]; in other words, only a tiny

[1] The OS-level malware will neither perform DoS attacks nor block/ delay regular communications and I/Os to avoid being noticed by the user (Sec. 3.2), and hence the CA will always provide this proxy service correctly.

[2] To prevent the adversary from guessing the secret message, we should use enough bits to represent it, and 100+ bits should be secure enough.

portion (e.g., less than 1%) of each write quest has been changed, creating a message which hides the secret message (we call this newly generated message the "steg-message"). This tiny modification will not be alerted by the adversary. 2) The write requests originally issued by the TApp on the external storage are always "cover messages" and allow being altered. 3) The adversary will not be able to access the TrustZone secure world, and hence will not be able to check out the original cover messages. Therefore, the adversary is not able to detect the existence of steganography by comparing the steg-messages with the corresponding cover messages.

Second, how can the MDetector securely convey a response back to the TApp? An immediate solution is that the MDetector modifies the data stored in a flash page to encode a secret response, and the TApp performs a read request on a location of the external storage which is mapped to this page. This immediate solution unfortunately is insecure, since the adversary can have access to the external storage beforehand, and obtain the original data stored in this page and, by comparing the original data with the data after being modified by the MDetector, the adversary can easily identify the existence of steganography. A key observation toward resolving this challenge is that, all the responses sent by the MDetector to the TApp are used to indicate a binary (0 or 1) result, e.g., the command (start or stop) is successfully performed (1) or not (0), or the malware is detected (1) or not (0). Specifically, after the TApp has sent a secret command to the MDetector by performing a write request (which stealthily encodes the secret command) on a random location, the TApp will immediately read the same location. If the read request is returned normally, it will indicate 0; otherwise, if the read request is returned with extra delay, it will indicate 1. Note that: 1) The extra delay should be a secret and dynamic value. The TApp can pick this delay value each time when sending a secret command, and treat it as a part of the secret command. 2) The extra delay can be plausibly denied as the normal system delay, since it happens rarely, e.g., it only happens when the MDetector starts to work, stops working, or has detected the malware.

4.2 Design Details

Let κ and l be security parameters. n is the size of a data block (in terms of bits), and s be the size of the messages (in terms of bits) communicated between the TApp and the MDetector. To avoid disturbing the regular workloads of the system, we reserve an area at the end of the block layer for the communication between TApp and the MDetector. The reserved area has $d \cdot N$ data blocks, with $d << 1$ and $d > 0$. Note that this reserved area does not lead to the compromise of steganography, as having a reserved area is pretty common, e.g., a swap space, a space for backup purpose, or a reserved space for a hidden volume [10,22]. We choose a pseudo-random function (PRF) f and a pseudo-random permutation π, defined as follows:
f: $\{0,1\}^{\kappa} \times \{0,1\}^{*} \rightarrow \{0,1\}^{l}$;
π: $\{0,1\}^{\kappa} \times \{0,1\}^{log_2^n} \rightarrow \{0,1\}^{log_2^n}$.

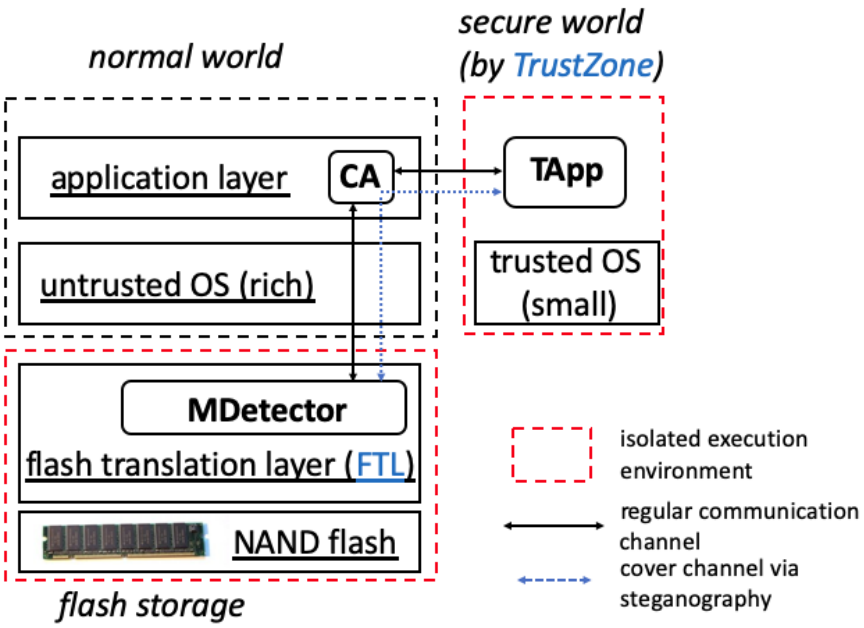

Fig. 2. The design overview of mobiDOM

Cover channel between TApp and MDetector via Steganography. The TApp and the MDetector independently maintain 2 secret keys k_1 and k_2, as well as a counter which will be initialized as 0 during system initialization. We define three secret messages[3]: *START*, *STOP*, and *QUERY*, which are communicated stealthily between the TApp and the MDetector via the steganography:

- START: The TApp uses this message to inform the MDetector to start the malware detection process. Note that the TApp will ask the CA to perform a regular write of DATA on a chosen data block of the block layer, and the DATA conceals the "START" message.
- STOP: The TApp uses this message to inform the MDetector to stop the malware detection process. Similarly, the TApp will ask the CA to perform a regular write of DATA which conceals the "STOP" message.
- QUERY: Periodically, the TApp sends a query (malware is detected or not) to the MDetector. The TApp will ask the CA to perform a regular write of DATA which conceals the "QUERY" message.

Note that each secret message is a collection of s bits determined during the initialization which is only known to the TApp and the MDetector, and s should be large enough to prevent brute-force attacks.

[3] mobiDOM only defines three basic messages to enable the basic functionality, but it can be easily extended to support extra communicated messages.

To start the malware detection, the TApp works as follows: It will generate DATA, pick a delay value, and encode the START message and the *delay* value into it, by running the Encode algorithm (Algorithm 1) using $START||delay$, DATA, k_1, counter as input, generating the stegDATA (corresponding to the "steg-message"). The TApp then runs Algorithm 3 using key k_2, counter, N as input, generating a random location j. The TApp further asks the CA to perform a write of stegDATA on block location j. The MDetector works as follows: It will monitor the write on the page corresponding to data block j (by applying Algorithm 3 using key k_2, counter, N as input, the MDetector can generate j), obtaining the stegDATA. It will then decode the stegDATA, by applying the Decode algorithm (Algorithm 2) using key k_1 and counter as input. It will further check whether the resulting decoded message contains the START message or not. If START is found, it will extract the *delay* value from the decoded message, start the malware detection, and add an artificial delay (determined by the extracted *delay* value) to the read request on the flash page corresponding to data block j. After having measured the read delay on the data block j, the TApp knows that the malware detection has been started successfully. Both the TApp and the MDetector will increase the stored counter by 1. To stop the malware detection, a similar process in both the TApp and the MDetector will be followed except that the secret message is replaced by STOP. Periodically, the TApp checks with the MDetector whether the malware has been detected or not using the QUERY message. The time interval for the this periodical check is a trade-off between the performance overhead and the delay of malware detection. In addition, the process for each check in both the TApp and the MDetector is similar to that for starting the malware detection, except that: 1) If no malware has been detected, the MDetector will not add an artificial delay to the read request on the flash page corresponding to data block j. This is advantageous, since the detected malware is a rarely happening event and most queries will not result in read delay. 2) Regardless whether the malware is detected or not, the counter in both the TApp and the MDetector will be increased by 1. In this way, the secret QUERY message will embedded differently each time, preventing the adversary from noticing the existence of steganography by comparing multiple subsequent write requests.

Algorithm 1: Encode

Input: MESSAGE, DATA, key k, counter
Output: stegDATA
View MESSAGE as a collection of t bits
View DATA as a collection of n bits
for $i = 1 : t$ **do**
  $j = \pi_k(counter||i)$
  $DATA[j-1] = MESSAGE[i-1]$
return DATA

Algorithm 2: Decode

Input: stegDATA, key k, counter
Output: MESSAGE
View stegDATA as a collection of n bits
for $i = 1 : t$ **do**
  $j = \pi_k(counter||i)$
  $MESSAGE[i-1] = stegDATA[j-1]$
return MESSAGE

Algorithm 3: Generate a random location in the reserved area

Input: key k, counter, N, d
Output: a random location in the reserved area
$j = (1-d) \cdot N + f_k(counter)\%(dN)$ **return** j

Malware detection in the FTL. The idea of our FTL-based malware detector (MDetector) is to detect the malware in the FTL by analyzing the access on the flash storage caused by software (malicious or not) running at the upper layers (i.e., the application or the OS layer). The key observation is that, the malware running at the upper layers may exhibit some unique access behaviors in the FTL [29] and, by capturing those behaviors, we may detect the malware. The advantage of MDetector is clear, since even if the malware can compromise the OS, it cannot compromise our MDetector which has been isolated from the OS at the hardware level. To function correctly, the MDetector relies on a classifier, which is able to classify any software as malicious and non-malicious in real time. The classifier should be trained using the set of pre-collected malware. Once the classifier has been trained and loaded, MDetector will monitor all the I/O requests issued from the upper layers, analyze them in real time, and decide whether there is malware present. Once any malware is identified, the MDetector will work with the TApp to get the user aware of the instance.

5 Discussion and Analysis

5.1 Discussion

Security of TrustZone. mobiDOM relies on an implied assumption that TrustZone itself is secure. This seems to be a widely acceptable assumption in the domain of TrustZone-based solutions [13]. There have been various attacks against TrustZone however, e.g., side-channel attacks [9, 20], CLKSCREW attacks [25], hardware-fault injection attacks [19], etc. Enhancing security of TrustZone has been actively taken care of in the literature [27] and is not the focus of this work.
Defending against other types of OS-level malware. Malware detection in the mobile devices has been a very challenging task because of the heterogeneity and diversity of the malware in the wild. This turns out to be even more challenging when the malware can obtain the OS-level privilege. mobiDOM can only

defend against a special type of OS-level malware which causes abnormal I/Os on the external storage media. For other types of OS-level malware, a potential solution is to run the malware detector in an isolated execution environment, which will then access the main memory of the normal world periodically (e.g., via direct memory access). We will further explore this in the future work.

Towards making mobiDOM more practical. One practical issue is to keep the FTL lightweight, since it is a thin firmware layer managed by less powerful processors and RAM. When integrating the malware detector into the FTL, a significant concern lies in the performance of ML-based detection. An option towards improving the performance would be conducting a model prunning [31], which can help increase inference speed and decrease storage size of the ML model; additionally, upon initializing mobiDOM, the ML model can be loaded into the RAM for malware detection later. Another practical issue is to manage interference of I/Os among regular software as well as multiple malware, which may perform I/Os simultaneously. Our experimental results in Sec. 6 only capture the scenario in which there is only one piece of malware running in the system. We will further investigate such interference in our future work.

5.2 Security Analysis

There are 3 major components in mobiDOM: the TApp, the MDetector and the communication messages between the TApp and the MDetector (Via the CA).

The security of the TApp is ensured by ARM TrustZone secure world. Without being able to compromise the TrustZone, the adversary is not able to compromise the TApp even if it can compromise the OS. In addition, the adversary will not be able to identify the existence of the TApp in the TrustZone secure world, since the TApp simply writes/ reads data to/ from the external flash storage (via the CA), which is pretty regular for any trusted applications running in the TrustZone.

The security of the MDetector is ensured by the FTL. Due to the isolation of the FTL in the hardware level, the adversary will not be able to compromise the FTL, even if it can compromise the OS. Therefore, the malware will not be able to notice the existence of MDetector, let alone to compromise it.

The communication messages between the TApp and the MDetector can be protected as analyzed in the following. First, their confidentiality can be ensured by staying hidden among regular I/Os via steganography. Specifically, the secret command messages from the TApp to the MDetector (START, STOP, QUERY) are hidden in the regular write requests on the external storage, which will be invisible to and hence unnoticeable by the adversary. In addition, the response messages are conveyed back from the MDetector to the TApp via read delays and, since the delay time is a one-time secret value, the adversary cannot interpret anything from such delays, which can be plausibly denied as occasional system delays (considering the delays in mobiDOM only happen when starting/ stopping the malware or having detected the malware). In addition, the existence of CA will not give the adversary any advantage of inferring the existence of mobiDOM since it is pretty common for the TrustZone-based applications to have CAs

running in the normal world to communicate with TAs located in the TrustZone secure world. Second, the integrity of the communication messages can be ensured. If the adversary modifies or replays the messages sent from the TApp to the MDetector, the MDetector can easily detect it since the secret command messages cannot be extracted successfully; if the adversary delays the communication messages sent from the TApp to the MDetector or the read responses from the MDetector to the TApp, the TApp can notice it considering the actual delay time in mobiDOM is a one-time secret value. Note that we do not consider DoS attacks in which the adversary blocks the communication messages or I/Os.

6 Implementation and Experimental Evaluation

To construct a mobile computing device following the architecture in Figure 1, we used two electronic development boards to build the testbed: 1) a Raspberry Pi [4] (version 3 Model B, with Quad Core 1.2GHz Broadcom BCM2837 64bit CPU, 1GB RAM) which is used as the host computing device, and 2) a USB header development prototype board LPC-H3131 [17] (with ARM9 32-bit ARM926EJ-S 180Mhz, 32MB SDRAM, and 512MB NAND flash) which is used as the external flash storage. The LPC-H3131 is connected to the Raspberry Pi via a USB2.0 interface (Figure 3). We have ported OP-TEE (Open Portable Trusted Execution Environment) [3] to the Raspberry Pi to facilitate the development of TrustZone applications. The TApp has been implemented into the TrustZone secure world as a trusted application (TA). In addition, we have ported [26] an open-source flash controller (FTL) OpenNFM [11] to LPC-H3131 and, after OpenNFM has been ported, the LPC-H3131 can be used as a flash-based block device by the host computing device via the USB2.0 interface. We have modified OpenNFM to support the communication between the TApp and the MDetector via steganography. In addition, a client application (CA) has also been built which runs in the normal world as a proxy for communication. For pseudo-random function, we used HMAC-SHA1, in which the size of the output hash value is 160-bit (i.e., l) and the key size is 128-bit (i.e., κ). The size of the secret message is 128-bit (i.e., s). N is approximately 250,000 (in terms of 2KB data blocks). To further optimize[4] the performance in TrustZone secure world a bit, we have pre-computed the PRF/ PRP values and stored them in the memory of the secure world during initialization.

Evaluating the communication between the TApp and the MDetector. To assess the time required to execute a command (START, STOP, QUERY) issued by the TApp to the MDetector, we have evaluated four cases: 1) no extra delay is added; and 2) 1-second delay is added; and 3) 3-second delay is added; and 5-second delay is added. We have measured 20 times for each case. Note that the

[4] Note that currently we have successfully created the testbed using the Raspberry Pi which: 1) successfully connects to our LPC-H3131 via USB2.0 and, 2) supports Arm TrustZone. However, Raspberry Pi is very poor in performance as observed in our experiments. We are testing new electronic development boards including BD-SL-i.MX6 [1] to build a more powerful testbed.

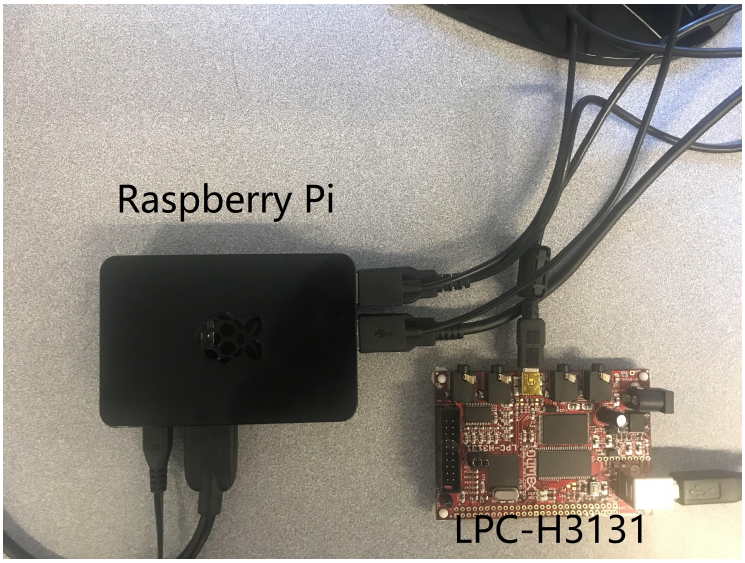

Fig. 3. The testbed for mobiDOM

performance of the Raspberry Pi will suffer due to temperature (known as thermal throttling) and, if the CPU temperature exceeds 60 Celsius (but should be below 85 degrees Celsius for it to work properly), the system will automatically throttle the processor. Our experiment results are shown in Figure 4. We can observe that: First, without adding any extra delay, it takes around 5 seconds to execute a command. The overhead mainly includes: encoding a secret message into the stegDATA (in the TrustZone secure world), writing the stegDATA to a flash page (in the normal world), extracting the secret message from the stegDATA (in the FTL), and reading the stegDATA from the flash page (in the normal world). A major time consumption comes from the Raspberry Pi since it significantly slows down when reaching 75 degrees Celsius. Note that this case is applied to three scenarios: 1) The START command is issued by the TApp, but the MDetector cannot successfully start the malware detection; 2) The STOP command is issued by the TApp, but the MDetector cannot successfully stop the malware detection; 3) The QUERY command is issued by the TApp, and no malware has been detected by the MDetector. Second, after adding different delays (1 second, 3 seconds, 5 seconds) in the FTL, the time required for executing a command can be easily differentiated from that no extra delay is added. This justifies the effectiveness of mobiDOM in conveying a response from the MDetector back to the TApp stealthily by adding extra delays. Note that this case is applied to three scenarios: 1) The START command is issued by the TApp, and the MDetector successfully starts the malware detection; 2) The STOP command is issued by the TApp, and the MDetector successfully stops

the malware detection; 3) The QUERY command is issued by the TApp, and the MDetector has detected some malware.

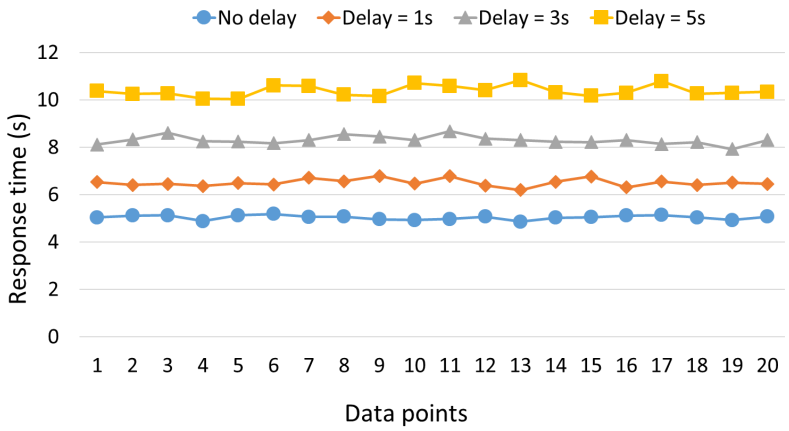

Fig. 4. Time for executing a command (START, QUERY, STOP) with/ without adding delays. The CPU temperature is around 75 degrees Celsius.

Evaluating the effectiveness of malware detection in the FTL. To understand the effectiveness of detecting malware in the FTL, we have collected 96 malware samples (mainly from VirusTotal [5]) and 36 benign software samples (including compression/ encryption/ deletion software, etc. which will cause I/Os to the external storage). For each sample, we manually ran it in a computer, which was connected to the LPC-H3131 via the USB, and collected the I/O traces (the entire dataset is available in [2]) in the FTL into a trace file. Note that after running each malware sample, we need to restore the entire system to the initial clean state. We used k-Nearest Neighbors (kNN), a supervised machine learning algorithm for classification. We chose k as 1. Our training set contains the I/O traces from 80 malware samples and 30 benign software samples. To test the effectiveness of the malware detection, we used the I/O traces from the remaining 16 malware samples and 6 benign software samples. For each trace file, we selected the first 30 I/O traces, which can be viewed as a three dimensional sequence. Also, we used the dynamic time warping method as the distance metric in kNN.

The experimental results are shown in Table 1. We can observe that: 1) The detection accuracy is 91%, which can justify the effectiveness of our FTL-based malware detector. 2) The false positive rate and the false negative rate are 33% and 0%, respectively. The low false negative rate indicates that our malware detector does not miss any malware if present. The false positive rate seems a little high in our malware detection. The reason is that, in our experiments, we have deliberately chosen those benign software samples which exhibit some

similar patterns to the collected malware. However, in practice, most of the benign software may not exhibit such similar patterns.

Accuracy	False negative rate	False positive rate
91%	0%	33%

Table 1. Detection results

7 Related Work

Aafer et al. [6] propose to detect Android malware by relying on critical API calls, their package level information, and their parameters. Zhang et al. [30] propose a semantic-based malware classification to accurately detect both zero-day malware and unknown variants of known malware families, in which they model program semantics with weighted contextual API dependency graphs. For ransomware, existing detection approaches mainly monitor typical file system activities [12, 14, 15, 21] or analyze cryptographic primitives [12, 15, 16]. For example, Unveil [14] generates an artificial user environment and monitors desktop lockers, file access patterns and I/O data entropy; CryptoDrop [21] observes file type changes and measures file modifications using a similarity-preserving hash function and Shannon entropy to detect ransomware. The aforementioned malware detection mechanisms work under the assumption that the malware cannot obtain the OS privilege, which is unfortunately not true for the OS-level malware.

MimosaFTL [28], SSD-Insider [7, 8], Amoeba [18] tried to detect the ransomware in the FTL. The major differences between them and mobiDOM are: First, mobiDOM is a general detection framework for any malware which causes abnormal I/Os on the external storage, rather than a specific framework for ransomware. Second, as a general malware detection framework, mobiDOM securely works with the user-level app, which is necessary for further actions after malware is detected. This is possible by leveraging both the ARM TrustZone (ensuring the security of the app) and the Steganography (ensuring the security of the communication between the malware detector and the user app) technique. On the contrary, the existing detection frameworks [7, 8, 18, 28] are specific for device-level data recovery and do not interact with user apps.

8 Conclusion

In this work, we have designed mobiDOM, a framework for combating the strong OS-level malware by smartly taking advantage of the existing secure features of mobile devices at the hardware level. Security analysis and experimental evaluation justify both the security and the effectiveness of mobiDOM.

9 Acknowledgment

This work was supported by US National Science Foundation under grant number 1938130-CNS, 1928349-CNS, and 2043022-DGE.

References

1. BD-SL-i.MX6. `https://boundarydevices.com/product/bd-sl-i-mx6/`.
2. Malware I/O Traces On Nand flash (MITON) V0.2. `https://snp.cs.mtu.edu/research/drm2/MITON-V0.2.zip`.
3. Open Portable Trusted Execution Environment. `https://www.op-tee.org/`.
4. Raspberry Pi 3 Model B. `https://www.raspberrypi.org/products/raspberry-pi-3-model-b/`.
5. VirusTotal. Retrieved May 17, 2019, from `https://www.virustotal.com/`.
6. Y. Aafer, W. Du, and H. Yin. Droidapiminer: Mining api-level features for robust malware detection in android. In *SecureComm*. Springer, 2013.
7. S. Baek, Y. Jung, A. Mohaisen, S. Lee, and D. Nyang. Ssd-insider: internal defense of solid-state drive against ransomware with perfect data recovery. In *ICDCS*. IEEE, 2018.
8. Sungha Baek, Youngdon Jung, Aziz Mohaisen, Sungjin Lee, and Daehun Nyang. Ssd-assisted ransomware detection and data recovery techniques. *IEEE Transactions on Computers*, 2020.
9. Sebanjila Kevin Bukasa, Ronan Lashermes, Hélène Le Bouder, Jean-Louis Lanet, and Axel Legay. How trustzone could be bypassed: Side-channel attacks on a modern system-on-chip. In *IFIP International Conference on Information Security Theory and Practice*, pages 93–109. Springer, 2017.
10. Bing Chang, Zhan Wang, Bo Chen, and Fengwei Zhang. Mobipluto: File system friendly deniable storage for mobile devices. In *Proceedings of the 31st Annual Computer Security Applications Conference*, pages 381–390. ACM, 2015.
11. Google Code. Opennfm. Retrieved May 17, 2019, from `https://code.google.com/p/opennfm/`, 2011.
12. Andrea Continella, Alessandro Guagnelli, Giovanni Zingaro, Giulio De Pasquale, Alessandro Barenghi, Stefano Zanero, and Federico Maggi. Shieldfs: a self-healing, ransomware-aware filesystem. In *Proceedings of the 32nd Annual Conference on Computer Security Applications*, pages 336–347. ACM, 2016.
13. Le Guan, Shijie Jia, Bo Chen, Fengwei Zhang, Bo Luo, Jingqiang Lin, Peng Liu, Xinyu Xing, and Luning Xia. Supporting transparent snapshot for bare-metal malware analysis on mobile devices. In *Proceedings of the 33rd Annual Computer Security Applications Conference*, pages 339–349. ACM, 2017.
14. Amin Kharraz, Sajjad Arshad, Collin Mulliner, William Robertson, and Engin Kirda. Unveil: A large-scale, automated approach to detecting ransomware. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 757–772, 2016.
15. Amin Kharraz, William Robertson, Davide Balzarotti, Leyla Bilge, and Engin Kirda. Cutting the gordian knot: A look under the hood of ransomware attacks. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 3–24. Springer, 2015.
16. Eugene Kolodenker, William Koch, Gianluca Stringhini, and Manuel Egele. Paybreak: defense against cryptographic ransomware. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, pages 599–611. ACM, 2017.

17. Mantech. Lpc-h3131. Retrieved May 17, 2019, from `https://www.olimex.com/Products/ARM/NXP/LPC-H3131/`, 2017.
18. D. Min, D. Park, J. Ahn, R. Walker, J. Lee, S. Park, and Y. Kim. Amoeba: an autonomous backup and recovery ssd for ransomware attack defense. *IEEE Computer Architecture Letters*, 17(2):245–248, 2018.
19. Pengfei Qiu, Dongsheng Wang, Yongqiang Lyu, and Gang Qu. Voltjockey: Breaching trustzone by software-controlled voltage manipulation over multi-core frequencies. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 195–209, 2019.
20. Keegan Ryan. Hardware-backed heist: extracting ecdsa keys from qualcomm's trustzone. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 181–194, 2019.
21. Nolen Scaife, Henry Carter, Patrick Traynor, and Kevin RB Butler. Cryptolock (and drop it): stopping ransomware attacks on user data. In *Distributed Computing Systems (ICDCS), 2016 IEEE 36th International Conference on*, pages 303–312. IEEE, 2016.
22. Adam Skillen and Mohammad Mannan. On implementing deniable storage encryption for mobile devices. In *20th Annual Network and Distributed System Security Symposium, NDSS 2013, San Diego, California, USA, February 24-27*, 2013.
23. Statista. Development of new android malware worldwide from june 2016 to march 2020. `https://www.statista.com/statistics/680705/global-android-malware-volume/`, 2020.
24. Kul Prasad Subedi, Daya Ram Budhathoki, Bo Chen, and Dipankar Dasgupta. Rds3: Ransomware defense strategy by using stealthily spare space. In *Computational Intelligence (SSCI), 2017 IEEE Symposium Series on*. IEEE, 2017.
25. Adrian Tang, Simha Sethumadhavan, and Salvatore Stolfo. {CLKSCREW}: exposing the perils of security-oblivious energy management. In *26th {USENIX} Security Symposium ({USENIX} Security 17)*, pages 1057–1074, 2017.
26. Deepthi Tankasala, Niusen Chen, and Bo Chen. A step-by-step guideline for creating a testbed for flash memory research via lpc-h3131 and opennfm. 2020.
27. Shengye Wan, Mingshen Sun, Kun Sun, Ning Zhang, and Xu He. Rustee: Developing memory-safe arm trustzone applications. In *Annual Computer Security Applications Conference*, pages 442–453, 2020.
28. P. Wang, S. Jia, B. Chen, L. Xia, and P. Liu. Mimosaftl: Adding secure and practical ransomware defense strategy to flash translation layer. In *CODASPY*. ACM, 2019.
29. Wen Xie, Niusen Chen, and Bo Chen. Poster: Incorporating malware detection into flash translation layer.
30. M. Zhang, Y. Duan, H. Yin, and Z. Zhao. Semantics-aware android malware classification using weighted contextual api dependency graphs. In *CCS*. ACM, 2014.
31. Michael Zhu and Suyog Gupta. To prune, or not to prune: exploring the efficacy of pruning for model compression. *arXiv preprint arXiv:1710.01878*, 2017.