Journal of Cybersecurity and Privacy

MDPI

Article

Defending against OS-Level Malware in Mobile Devices via Real-Time Malware Detection and Storage Restoration †

Niusen Chen and Bo Chen *

Department of Computer Science, Michigan Technological University, Houghton, MI 49931, USA; niusenc@mtu.edu

* Correspondence: bchen@mtu.edu; Tel.: +1-906-487-3149

† This manuscript is an extended version of our conference paper published in Applied Cryptography and Network Security Workshops, Kamakura, Japan, 21–24 June 2021.

Abstract: Combating the OS-level malware is a very challenging problem as this type of malware can compromise the operating system, obtaining the kernel privilege and subverting almost all the existing anti-malware tools. This work aims to address this problem in the context of mobile devices. As real-world malware is very heterogeneous, we narrow down the scope of our work by especially focusing on a special type of OS-level malware that always corrupts user data. We have designed mobiDOM, the first framework that can combat the OS-level data corruption malware for mobile computing devices. Our mobiDOM contains two components, a malware detector and a data repairer. The malware detector can securely and timely detect the presence of OS-level malware by fully utilizing the existing hardware features of a mobile device, namely, flash memory and Arm TrustZone. Specifically, we integrate the malware detection into the flash translation layer (FTL), a firmware layer embedded into the flash storage hardware, which is inaccessible to the OS; in addition, we run a trusted application in the Arm TrustZone secure world, which acts as a user-level manager of the malware detector. The FTL-based malware detection and the TrustZone-based manager can communicate with each other stealthily via steganography. The data repairer can allow restoring the external storage to a healthy historical state by taking advantage of the out-of-place-update feature of flash memory and our malware-aware garbage collection in the FTL. Security analysis and experimental evaluation on a real-world testbed confirm the effectiveness of mobiDOM.

Keywords: OS-level malware; data corruptions; detection; recovery; mobile devices; TrustZone; flash translation layer

Citation: Chen, N.; Chen, B. Defending against OS-Level Malware in Mobile Devices via Real-Time Malware Detection and Storage Restoration. *J. Cybersecur. Priv.* **2022**, *1*, 1–19. https://doi.org/

Academic Editor: Hossein Saiedian

Received: 10 April 2022
Accepted: 22 May 2022
Published:

1. Introduction

We have been witnessing a surge of malware on mobile devices in the past few years [1]. The malware intrudes into a victim's mobile device, stealing personally private or even mission-critical data, corrupting the local storage [2], or controlling the victim's entire device. In particular, there is one type of strong malware that is able to compromise the entire operating system of the device, obtaining the kernel privilege. This type of "OS-level" malware is extremely difficult to combat, since it can easily subvert any traditional anti-malware software or tools running in the user/kernel space [3–8] by leveraging its high privilege. It would be challenging or even impossible to design a common defense strategy that can combat any OS-level malware as the malware today is highly heterogeneous. In this work, we focus on a special type of OS-level malware that always corrupts data stored in the victim mobile device, i.e., the *data corruption malware*. This type of malware includes ransomware that encrypts the user data, making it inaccessible to the user, or any computer virus, trojan horse, etc., which first steals the user data by retrieving it from the victim device and sending it to a remote server, and then deleting it from the device. The data corruption malware we consider here is assumed to be able to obtain the kernel privilege by compromising the OS.

To combat the data corruption malware in the mobile devices, one strategy would be ensuring that the data being corrupted are always recoverable after the malware attack. To achieve this guarantee, our design contains two components: (1) a malware detection component (i.e., *malware detector*), which can detect the presence of data corruption malware timely, and (2) a data recovery component (i.e., *data repairer*), which can restore the external storage to a good historical state once the malware is detected and removed. The design rationale of each component is as follows:

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

Compared to traditional personal computers, mobile computing devices today are equipped with unique hardware features: (1) They usually use ARM processors that have reduced circuit complexity and low power consumption. ARM processors have an integrated *TrustZone*, which is a hardware security feature in any Cortex-A processor (built on the Armv7-A and Armv8-A architecture) and Cortex-M processor (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, and 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, our design of the malware detector is: first, we integrate the malware detection into the FTL. Such a design is advantageous because: (1) the OS-level malware will not be able to subvert the malware detection located in the FTL, which is isolated by the flash storage hardware and remains transparent to the OS; and (2) the data corruption malware usually needs to perform I/Os on the external storage, and such I/Os may exhibit some unique behaviors that can be observed in the FTL as confirmed in our experiments using real-world malware samples. Second, to prevent the user-level manager 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 manager is secure and is able to work with the malware detection in the FTL correctly. In addition, to prevent the malware from noticing the communication between the malware detection (in the FTL) and the manager (in the TrustZone), we leverage steganography so that the manager and the malware detection can communicate stealthily via the regular I/Os performed on the external storage.

Data repairer. To enable restoration of external storage after malware attacks, a key question would be finding out those original data before the malware corrupts them. Our observation is that: the flash memory performs an out-of-place update [9], and therefore, any delete/overwrite operations issued by the malware will not immediately delete data from the underlying flash memory; instead, the data will just be invalidated and removed later by a garbage collector running in the FTL. This observation makes it possible to enable data recovery after malware attacks by extracting the invalidated data from the raw flash memory. However, the garbage collector will eventually reclaim the flash memory space, which stores the invalidated data, and once this happens, the original data deleted by the

malware will be irrecoverable. To resolve this challenge, we take advantage of the designed malware detector. Specifically, we divide the time into epochs, and if the malware detector does not detect any malware by the end of an epoch, it will inform the garbage collector to reclaim the flash memory space that holds invalid data (i.e., the garbage collector will not reclaim the flash memory space until the end of each epoch and after being notified of no malware); if the malware detector detects any malware, the garbage collector will be frozen immediately, and the invalid data (which have not yet been deleted by the garbage collector) will be extracted from the flash memory to restore the external storage.

The resulted system design, mobiDOM, is the first full-fledged defense for mobile devices that can combat the data corruption malware even if the malware can compromise the entire OS. Compared to our conference version [10], the major differences are: (1) We have incorporated a malware-aware garbage collector that only reclaims flash memory space occupied by the invalid data if the malware is not detected. On the contrary, reference [10] uses the original garbage collector incorporated with the FTL, which may delete the data invalidated by the malware and, hence, cannot ensure data recovery after malware detection. (2) We have proposed the first complete framework that can defend against the OS-level data corruption malware. The framework includes both a malware detection and a data repairer component. Especially, the data repairer component allows restoring the external storage back to a good historical state after detecting the malware. On the contrary, reference [10] only supports malware detection, lacking the capability of data restoration. (3) We have provided experimental results for the newly added component.

Contributions. We made the following contributions in this work:

- We have proposed the first framework that can combat the OS-level data corruption malware for mobile computing devices. Our framework contains a malware detector and a data repairer.
- Our malware detector can securely detect the malware even if it can obtain the kernel privilege. This is feasible by utilizing both the isolation environments (i.e., Arm TrustZone and FTL) provided by mainstream mobile computing devices as well as a novel steganography technique.
- Our data repairer can restore the external storage to a healthy historical state. This is feasible by leveraging the out-of-place-update feature of flash memory and modifying the existing garbage collection in the FTL.
- We have analyzed the security of mobiDOM. In addition, we have implemented a prototype of mobiDOM and ported it to a real-world testbed to assess its performance.

2. Background

2.1. Flash Memory

Flash memory includes NAND flash and NOR flash. Especially NAND flash has been extensively used as mass storage (e.g., SD/miniSD/microSD cards, MMC cards, UFS cards) due to its cheap price. Compared to conventional hard disk drives (HDD), the flash memory is electrically erasable and re-programmable and does not rely on mechanical components. Therefore, it usually has much higher I/O throughput and much lower noise compared to HDDs. The NAND flash is organized into blocks, each of which is typically a few hundred kilobytes in size. A flash block consists of pages, each of which is a few kilobytes in size. Typically, the I/O operations are performed on the basis of pages, while the erase operations are performed on the basis of blocks.

Compared to traditional mechanical disk drives, NAND flash is different in nature: First, it uses an erase-then-write design, i.e., a flash memory cell needs to be erased first before it can be re-programmed. Therefore, to modify the data stored in a page, we need to erase the entire encompassing block. This, however, requires copying out valid data stored in this block, erasing the block, and writing the data back, leading to significant write amplification. To mitigate the write amplification, the flash memory typically uses an out-of-place update strategy (see Figure 1), i.e., when the OS performs an overwrite operation, the old data will be temporarily preserved in the flash memory, and the new data

will be written to new locations. The data repairer component of our design will utilize this feature to restore external storage to a good historical state after malware detection. 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 is not able to reliably store data anymore. Therefore, to prevent a single flash block from being erased/programmed too frequently, wear leveling is introduced to distribute P/Es evenly across the entire flash. Third, reading/writing a flash memory cell too frequently may flip the bits stored in its nearby cells of the same block, leading to read/write disturb errors.

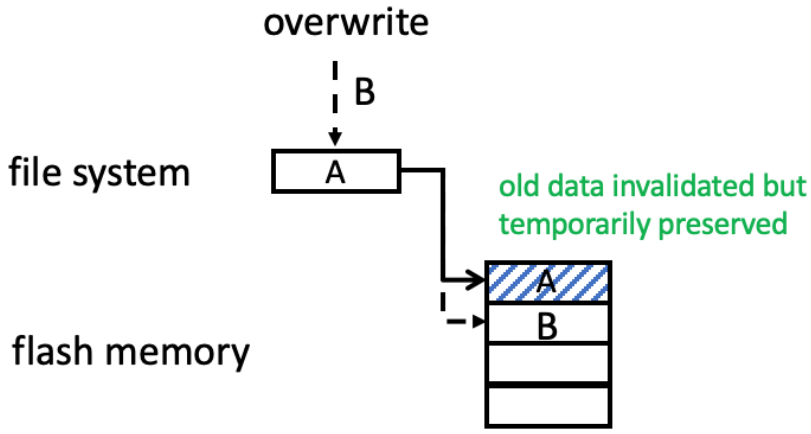

Figure 1. The out-of-place update in flash memory. When data A is overwritten by data B in the file system, data B will be placed on another flash page, and data A will be invalidated without actually being deleted from its holding flash page.

2.2. Flash Translation Layer (FTL)

To manage the special hardware nature of flash memory, a flash storage medium typically incorporates a flash translation layer (FTL), which can handle the unique characteristics of NAND flash internally, exposing a block access interface externally (Figure 2). Therefore, block-based file systems (e.g., EXT, FAT, NTFS, etc.) that have been designed for HDDs can be deployed on top of a flash storage medium. The FTL implements four key functions: address translation, garbage collection, wear leveling and bad block management. Address translation manages mappings between the block addresses and the flash memory addresses so that I/Os performed on the block addresses can be translated to I/Os performed on the flash memory. Due to the out-of-place update feature, the data stored in flash blocks may be invalidated over time. Garbage collection is used to periodically reclaim those invalid blocks. Wear leveling ensures that programs/erasures can be distributed evenly across the flash to prolong the service life of flash memory. Unavoidably, "bad" blocks will be generated over time, and bad block management is used to handle those bad blocks, preventing them from being used again.

The FTL stays inside a flash storage device and is isolated from any external entities (e.g., the OS) using the device. Leveraging this hardware-level isolation, our design will integrate the malware detection into the FTL. The benefit is it would not be possible for the malware to subvert our malware detection, even if the malware can compromise the OS.

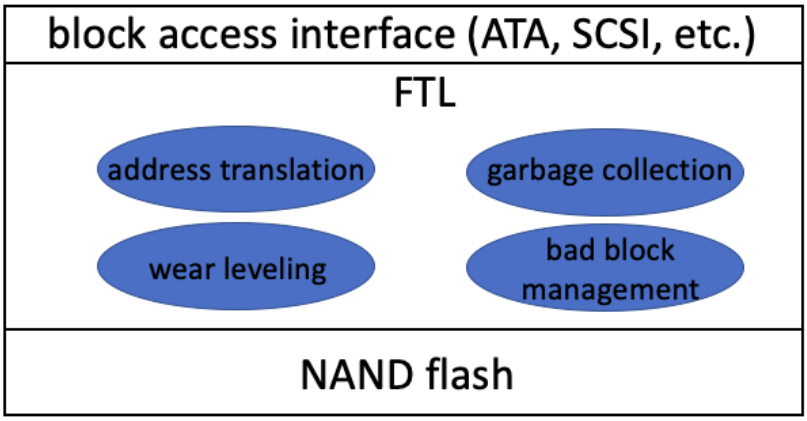

Figure 2. Flash translation layer (FTL).

2.3. ARM TrustZone

A trusted execution environment (TEE) is part of the main processor, which provides an isolated environment to protect both the confidentiality and the integrity of code/data loaded inside. TrustZone is a TEE implementation of Arm processors, 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). The idea of TrustZone is to create two execution environments that run simultaneously on a single processor: a secure execution environment (i.e., a *secure world*), which runs sensitive applications, and a non-secure execution environment (i.e., a *normal world*), which runs 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 will be aware of the corresponding world of the core, and applications running in the normal world cannot access the memory space of the secure world. Therefore, critical applications can be run in a secure world, and their confidentiality and integrity cannot be compromised even if the adversary can compromise the OS. Leveraging TrustZone, our design will run a user-level manager of the malware detector in the TrustZone secure world to avoid being compromised by the OS-level malware.

2.4. Steganography

Steganography allows a sender to send a seemingly innocuous cover message to a receiver that conceals some critical message. In this way, the critical message can be delivered to the receiver without the adversary being aware by staying in the middle. Compared to cryptography, which protects the content of a critical message (e.g., encryption transform the message to some random format based on secret keys so that the plaintext of the message can be concealed), steganography is more advantageous as it not only hides the content of the message but also essentially hides the fact that the message exists. Our design will leverage the steganography technique to hide the communication between the user-level manager running in the TrustZone secure world and the malware detector running in the FTL.

3. System and Adversarial Model

3.1. System Model

We consider a low-power mobile computing device that is equipped with an ARM processor (with TrustZone support) and a flash-based block device as external storage (e.g., a miniSD/microSD card, an eMMC card, or a UFS card). This type of mobile device can be found broadly in the real world, including smartphones, tablets, wearable devices, as well as the ever-growing Internet of Things (IoT) devices. A general architecture of this type of device is shown in Figure 3. Leveraging TrustZone, we can create a secure world that runs trusted applications (**TA**) on top of a small trusted OS and a normal world that runs

untrusted applications on top of a regular insecure OS. The flash-based block device has a built-in FTL, which transparently handles the unique nature of NAND flash and exposes a block access interface. We assume there are N data blocks on the block layer that are usable by the OS, and the size of a data block is assumed to be equal to the size of a flash page for simplicity. For example, if the flash page size is 2 KB, each data block can be viewed as a collection of 4 512-byte sectors.

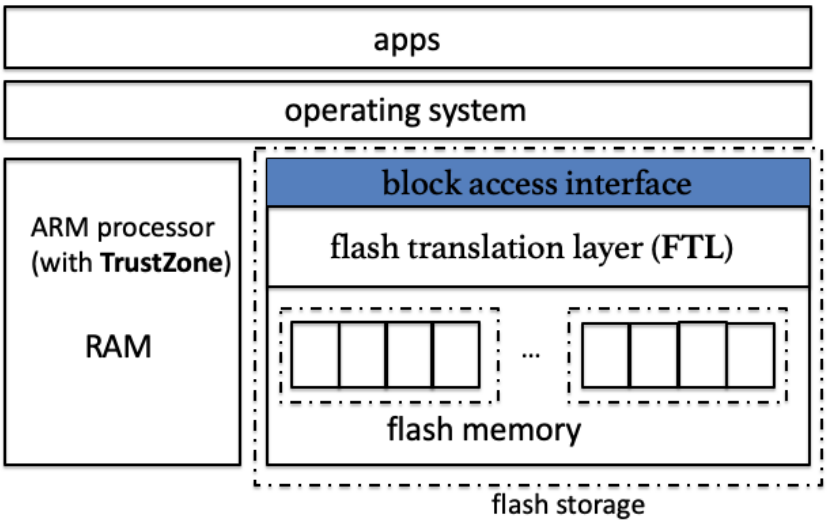

Figure 3. The architecture of a mobile device.

3.2. Adversarial Model

We consider high-privileged OS-level malware that is able to compromise (the malware usually starts from a user-level privilege and gradually escalates itself to the OS privilege by exploiting various system vulnerabilities.) the regular insecure operating system of a victim's mobile device. After having compromised the OS, the malware would be able to subvert any malware detection tools running in the insecure OS. Our design relies on a few assumptions: First, the malware is not able to compromise the trusted OS and the TAs running in the TrustZone secure world. This is a general assumption in the domain of TrustZone technologies [9]. The malware is not able to hack into the FTL. This is a reasonable assumption, as a flash storage device only provides limited access interfaces to the OS, and we have not been aware of any attacks that can take advantage of these interfaces to compromise the FTL. Second, we assume that the malware will not perform DoS attacks. Specifically, it will neither block regular (or seemingly regular) communications between the TAs running in the TrustZone secure world and the applications running in the normal world, nor block I/Os performed by TAs on the external storage. This assumption is reasonable, as otherwise, the TAs in the secure world can easily detect such blocking and inform the user of the potential presence of malware. However, the malware may try to view, modify or replay the communicated messages without being detected. Third, we consider data corruption malware, which always needs to perform I/Os on the external storage. This type of malware is commonly found in practice, e.g., a piece of ransomware that encrypts user data and asks for ransom or a trojan which first steals user data, and then corrupts them locally.

4. mobiDOM Design

mobiDOM aims to combat the data corruption malware that can compromise the operating system of a victim's mobile device. Our key idea towards the defense is to ensure that the external storage corrupted by the malware can be restored after the malware attacks. Therefore, mobiDOM continuously monitors the system in real-time, and once the presence of the malware is detected, the user will be informed (step 1); then, the malware will be

quarantined and removed by the user (step 2); last, the external storage will be restored (step 3). Step 2 is pretty common in any anti-virus or anti-malware tools, and therefore, mobiDOM will focus on step 1 and step 3. Correspondingly, mobiDOM mainly consists of a **malware detector**, which is responsible for detecting the malware and informing the user, and a **data repairer**, which is responsible for restoring the external storage to a good historical state after the malware is detected and removed. The design of our malware detector and our data repairer will be elaborated on in Sections 4.1 and 4.2, respectively.

4.1. Malware Detector

Detecting the OS-level malware is a very challenging task, as the malware can compromise the operating system and subvert any malware detectors running at either the user level or the kernel level. To prevent the malware detector from being compromised by the malware, our key idea is to run it in an environment isolated from the OS. In this way, even if the OS is compromised, the malware detector can still function correctly. We introduce TApp, a trusted application running in the TrustZone secure world, and MDetector, a malware detector running in the FTL that has been isolated from the OS. The TApp acts as a trusted controller of the MDetector. To facilitate the communication between the TApp and the MDetector, we introduce a client application (CA) in the normal world, which acts as a communication proxy. Note that: (1) The OS-level malware will not block regular communications and I/Os to avoid being aware by the user (Section 3.2), and therefore, the CA will always provide this proxy service correctly. (2) Enabling the TApp to directly read/write the external flash storage via the trusted OS running in the TrustZone is not a good alternative, as it requires adding a lot of extra software components (i.e., disk drivers and other necessary components in the storage path) to the small trusted OS, imposing a lot of extra burden on the small trusted OS. Furthermore, accessing the external storage directly in the TrustZone secure world is rarely supported [11]. As the communication proxy (i.e., the CA) is running in the insecure normal world, the malware may detect the communications between the TApp and the MDetector and can simply disturb their communications (Section 3.2). We, therefore, utilize steganography so that the CA and the MDetector can communicate stealthily without being detected by the malware. An overview of our designed malware detector is shown in Figure 4.

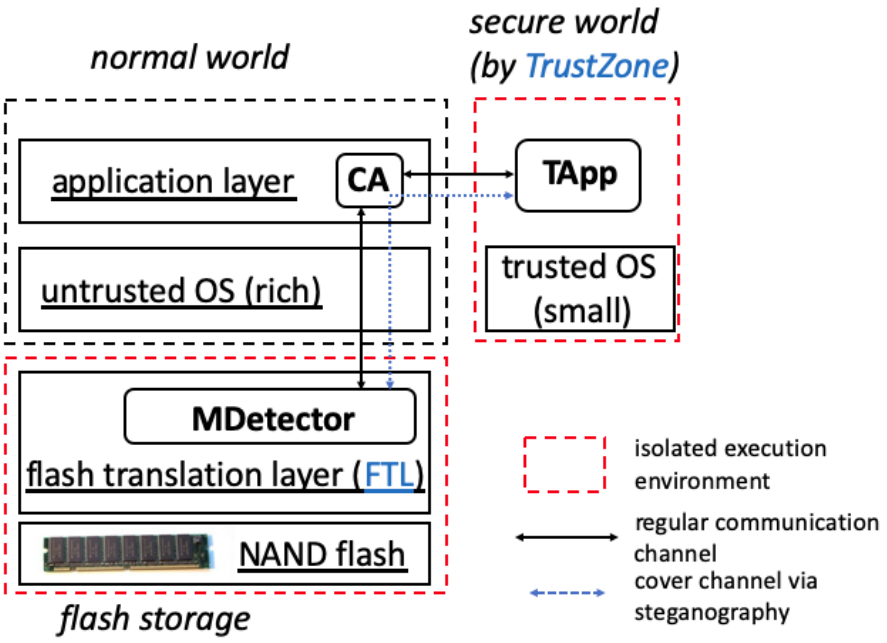

Figure 4. The design overview of our malware detector.

4.1.1. Enabling Stealthy Communications between The TApp and MDetector

The TApp and the MDetector will communicate with each other through the CA stealthily. In particular, the TApp will issue commands, and the MDetector will execute the commands and send back responses. For simplicity, we define three basic commands for the TApp to control the MDetector:

- *START*: This command will request the MDetector to start the malware detection process.
- *STOP*: This command will request the MDetector to stop the malware detection process.
- *QUERY*: This command is used by the TApp to check whether there is any malware detected by the MDetector. It will be performed by the TApp periodically.

Note that: (1) Each command 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, e.g., 128 bits. (2) Our design can be easily extended to support extra commands.

To send a command stealthily to the MDetector, the TApp will leverage steganography. The TApp and the MDetector will share a secret key during the initialization. Each time upon sending a command, the TApp will use a pseudo-random function to select a random portion of bits from a cover message based on the secret key, encode the command into the selected bits (i.e., we simply replace the selected bits with the bits corresponding to a command), and convey the modified cover message (denoted as stegDATA) to the MDetector through a regular write request. The write request will be forwarded to the flash storage medium via the CA. Upon receiving the write request, the MDetector can extract the command from the stegDATA via the secret key. By observing the stegDATA, the adversary will not be able to know there is a secret command embedded because: (1) The adversary cannot compromise the TrustZone secure world and does not have access to both the original cover message and the encoding process. (2) Compared to the size of a regular write request (typically a few KBs), the size of the command is small (only 100+ bits). It is unlikely for the adversary to be aware of such a tiny modification of the original cover message. A concrete example of sending a secret command from the TApp to the MDetector is provided in Figure 5.

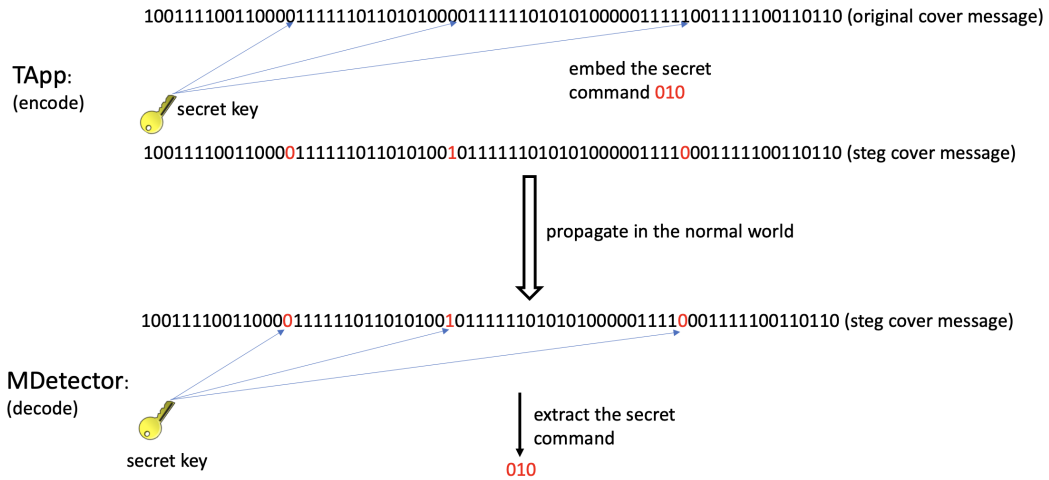

Figure 5. A concrete example of propagating a secret command 010 via steganography.

Each time after finishing handling a command, the MDetector should send back a response to the TApp stealthily. Each response is usually a binary value. For example, 1 indicates that the START or STOP command is successfully executed, while 0 indicates a failure; 1 indicates that the malware is detected, while 0 indicates no detection has been detected. To send back such a binary response in a stealthy manner, our strategy is to utilize

the delay of read requests performed on the flash storage. Specifically, after the TApp sends the secret command to the MDetector by encoding it into a regular write request on a disk location, the TApp will read the same location immediately. A normal return without extra delay will indicate value 0 while a return with artificial delay will indicate value 1. Note that: (1) For the security concern, the delay value should be a secret and dynamic value. It can be set by the TApp when sending a secret command each time, and this value can also be encoded into the cover message stealthily. (2) The artificial delay should be small enough to avoid being noticed by the adversary. In addition, the artificial delay will be added only after the MDetector has successfully started/stopped the malware detection process or detected the malware. All of them happen rarely and can be reasonably denied as the system delay.

4.1.2. Real-Time Malware Detection in The FTL

The MDetector is implemented in the flash translation layer so that even if the malware can compromise the operating system, it cannot subvert the MDetector. We only consider the data corruption malware (Section 3.2), which will always create issues with I/Os on the external storage in order to corrupt user data. Our intuition is that special behaviors of malware at upper layers will eventually cause special access patterns [12] on underlying flash memory, and by capturing and extracting those patterns in the FTL, it is possible to detect the malware. To function correctly, the MDetector relies on a classifier, which is able to classify any I/Os as malicious and non-malicious in real-time. The classifier can be trained using the I/Os generated by running a set of pre-collected malware (Section 6.1). 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. If any malware is detected, the MDetector will work with the TApp to get the user aware of the instance (i.e., via the QUERY command issued by the TApp periodically).

The design details of the classifier. Each malware/software may generate a sequence of I/Os in the FTL. Each I/O typically contains the following information: (1) a flag indicating whether this I/O is corresponding to a read or a write operation, (2) the address of this I/O, and (3) the data length of this I/O. Our objective is to differentiate the I/O sequence of the malware from that of the regular software. To achieve this objective, we use k-Nearest Neighbors [13] (kNN), a non-parametric supervised learning algorithm. We choose the sequence of fixed-length I/Os captured for each software/malware sample as the feature. An important step for kNN is to measure the distance between any two software/malware samples. However, it is challenging to measure the distance between two I/O sequences. To address this challenge, we utilize the dynamic time warping algorithm [14], which has been designed to measure the similarity between two temporal sequences. Given two sequences S_1 and S_2, we compute its distance following these steps:

1. We view each sequence as a collection of points, and each point corresponds to an I/O of this sequence. We also initialize empty queues Q.
2. For S_1, we compute the euclidean distance from its first point to each of the points in S_2. The minimal distance will be used as the distance between the first point to S_2 and added to Q.
3. We repeat Step 2 for each of the remaining points in S_1.
4. For S_2, we compute the minimal distance between each of its points to S_1 by repeating Step 2 and 3.
5. The distance between S_1 and S_2 will be the sum of all the values in Q.

4.2. Data Repairer

The OS-level malware may corrupt data stored in the victims mobile device. For example, it can overwrite the local data with garbage data or simply delete them. However, the delete operation [15] in the system level usually only leads to the deletion of the corresponding metadata in the file system and the actual data are not deleted. Therefore, a reliable manner for the malware to corrupt data would be overwriting the data with

something else. Towards data restoration after malware attacks, our key observation is that the mobile device usually uses flash storage, which follows a unique out-of-place update strategy; due to the out-of-place update, the original data corrupted by the malware would be temporarily preserved in the flash memory and could be extracted for data recovery after attacks. Especially, the FTL usually maintains a table keeping track of mappings between data stored in the flash memory and that viewed by the OS. For example, in Figure 6a, for data D, the FTL maintains a mapping p between its address in the OS and its address in the flash memory. After the malware corrupts data D by overwriting it with garbage data D', the garbage data D' will be stored in a new flash memory address due to the out-of-place update feature, and the FTL will change the mapping from p to $p^{'}$. Therefore, if we can restore the current mapping $p^{'}$ to the old mapping p, the data "corrupted" by the malware can be restored.

There are three issues that need to be addressed toward a robust data recovery after the malware attack: (1) After the malware corrupts data at the OS level, the data (e.g., D) stored at the flash blocks will be invalidated, and the garbage collection (GC) [16,17] may reclaim those blocks, rendering the data irrecoverable. (2) The old mapping p may be deleted by the FTL before the data recovery is performed. (3) The user stays at the user level, and may not be able to control the data repairer upon data recovery. In the following, we will discuss our strategies to address each of the three issues.

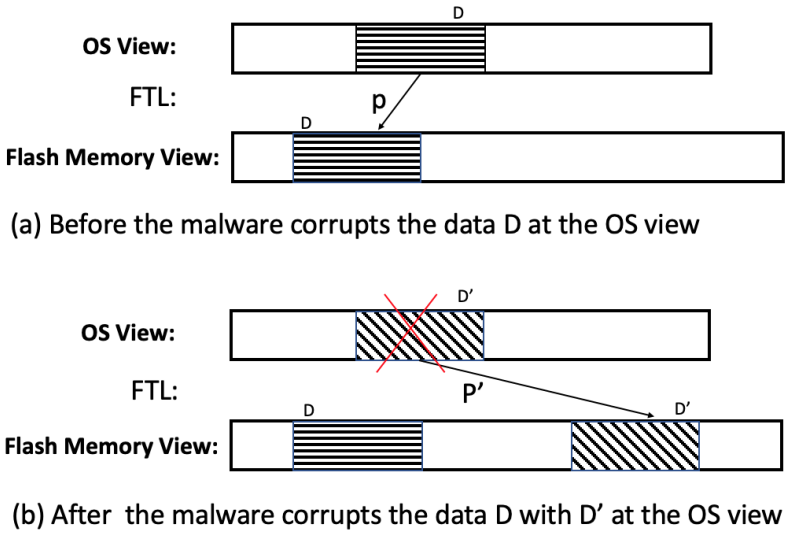

Figure 6. Mappings between the OS view and the flash memory view.

Malware detection-aware garbage collection. To prevent the GC from reclaiming those flash blocks that store the data invalidated by the malware, we utilize a malware detection-aware GC strategy (Figure 7). Especially, we divide the time into epochs, and the GC will be performed periodically at the end of each epoch; before the GC functions and reclaims flash blocks filled with invalid data, it will first check with the malware detector, and the flash blocks will be reclaimed only if there is no malware detected at this point (otherwise, the GC will be frozen until the data recovery is invoked and successfully conducted). Another issue is that when the malware starts to function, the malware detector may not be able to detect it instantly. Therefore, there is a potential detection delay. To compensate this delay, the GC on a flash block should be delayed accordingly. Fortunately, in our malware detection experiments (Section 6.1), the detector can detect the malware after around 30 I/Os, which indicates that the delay is very subtle.

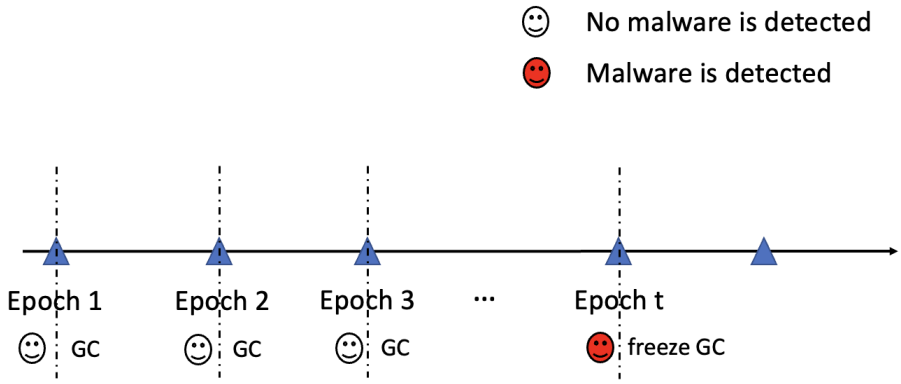

Figure 7. Malware detection-aware garbage collection.

Ensuring recoverability of the old mappings. As the flash storage performs the out-of-place update, an immediate strategy is to prevent the GC from reclaiming flash blocks storing invalid mappings so that after the malware attacks, the old mapping p can be extracted from the flash memory. This solution ensures the recoverability of old mappings, but localizing the old mappings corresponding to a certain historical version would be difficult. In addition, retaining all the historical mappings may incur a large storage overhead. Another strategy is to back up the most recent version of the mappings at the end of an epoch if the malware detector does not detect any malware (Figure 8). In this way, upon detecting any malware and restoring the system, the most recent version of the mappings can be retrieved for data recovery. Note that: (1) The flash blocks that store the backup of mappings should be invisible to the upper layers. (2) Only the most recent version of the mappings is needed to be backed up (in case there is any malware detection delay, storing 2–3 most recent versions is always preferred, which does not require too much extra storage), which does not incur a large storage overhead as the size of mappings is significantly smaller than that of the data.

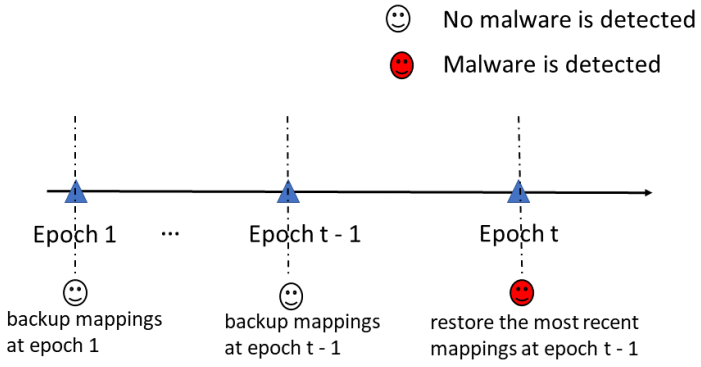

Figure 8. Mappings backup and restoration.

Invoking the data repairer at the user level. We define a new command REPAIR, which is a collection of s bits determined during the initialization and is only known to the user and the data repairer in the FTL. To invoke the data repairer after the malware has been detected and cleared, the user can write the REPAIR command to a reserved block

address. The data repairer, which is running in the FTL, will monitor the reserved block address, and once the REPAIR command is received, it will start to restore the data. Note that: (1) as the data repairer is invoked only after the malware has been removed, the user can simply stay in the normal world. (2) The REPAIR is a secret command with s bits (s is typically large enough); therefore, the malware, as well as regular applications will not be able to invoke the data repairer accidentally.

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 [9]. However, there have been various attacks against TrustZone, e.g., side-channel attacks [18,19], CLKSCREW attacks [20], hardware-fault injection attacks [21], etc. Enhancing the security of TrustZone has been actively taken care of in the literature [22] and is not the focus of this work.

Defending against other types of OS-level malware. mobiDOM can only combat the OS-level data corruption malware that causes I/Os to the external storage. 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) in order to identify any misbehavior.

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 for improving the performance would be conducting a model pruning [23], which can help increase inference speed and decrease the 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 Section 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.

Handling false positives and false negatives of the malware detector. Malware detection relies on the AI model, which typically suffers from false positives and false negatives. If there is a false positive, the user will be falsely notified of a malware detected event. To mitigate this false alarm, the user is recommended to double confirm the attacks manually before further action is taken. If there is a false negative, the GC may have reclaimed some of the invalid flash blocks that store original data. Fortunately, our experimental results (Section 6.1) show that the false negatives are rare. The false negative issue will be investigated further in our future work.

Malware detection delay. Our malware detection aware GC strategy relies on the timely detection of the malware. An extreme case is when the malware starts to compromise data, but the malware detector has not yet detected it due to the detection delay. This is right at the end of an epoch, and the GC will check with the malware detector. As no malware has been detected yet, the GC will reclaim invalid blocks that have been just invalidated by the malware. A remediation strategy is that the GC should avoid reclaiming those flash blocks that have been invalidated recently. This will also help mitigate the false negative issue mentioned before.

5.2. Security Analysis

mobiDOM contains both a malware detector and a data repairer component. The data repairer component does not have any security concerns as it runs after the malware has been detected and removed. Therefore, we focus on the security analysis over the malware detector component in the following.

There are three major components in the malware detector: 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 at 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 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. Experimental Evaluation

To construct a mobile computing device following the architecture in Figure 3, we used two electronic development boards to build the testbed: (1) a Raspberry Pi [24] (version 3 Model B, with Quad Core 1.2 GHz Broadcom BCM2837 64 bit CPU, 1 GB RAM), which is used as the host computing device, and (2) a USB header development prototype board LPC-H3131 [25] (with ARM9 32-bit ARM926EJ-S 180 Mhz, 32 MB SDRAM, and 512 MB NAND flash), which is used as the external flash storage. The LPC-H3131 is connected to the Raspberry Pi via a USB2.0 interface. We have ported OP-TEE (Open Portable Trusted Execution Environment) [11] 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 [27] 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 that runs in the normal world as a proxy for communication. For a pseudo-random function, we used HMAC-SHA1, in which the size of the output hash value is 160-bit, and the key size is 128-bit. The size of a command is 128-bit.

We also modified the OpenNFM so that after the malware has been detected, the external storage can be restored to a good historical state. Periodically at the end of each epoch, the flash controller retrieves the current version of the mapping table, and stores it elsewhere in the flash memory if there is no malware found by the MDetector. Note that

the garbage collection will be performed at the end of each epoch only when no malware is detected at that point. After the MDetector has detected the malware, the user will be aware of this attack by issuing a QUERY message via the TApp. The user can then block and remove the malware. Finally, the user will invoke the data repairer in the flash controller by issuing the REPAIR message on the reserved block address. Note that the malware has been eliminated at this point, and the user can invoke the data repairer in the normal world. The flash controller will restore the mapping table by retrieving the most recent mapping table from the flash memory.

6.1. Experimental Results for The Malware Detector

Evaluating the communication between the TApp **and the** MDetector. We have tested four different cases to evaluate the communication between TApp and MDetector: (1) no extra delay is added; (2) 2-s delay is added; (3) 3-s delay is added; (4) 5-s delay is added. We have collected 20 data points and the results are shown in Figure 9. We have the following observations: (1) Without additional delay, 5 s are needed to execute a command with normal return. The overhead mainly includes: encoding a secret command into the stegDATA (in the TrustZone secure world), writing the stegDATA to a flash page (in the normal world), extracting the secret command 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 °C*.(2) With the additional delay, TApp can successfully differentiate the responses from the normal return. This justifies the effectiveness of mobiDOM in conveying a response from the MDetector back to the TApp stealthily by adding extra delays.

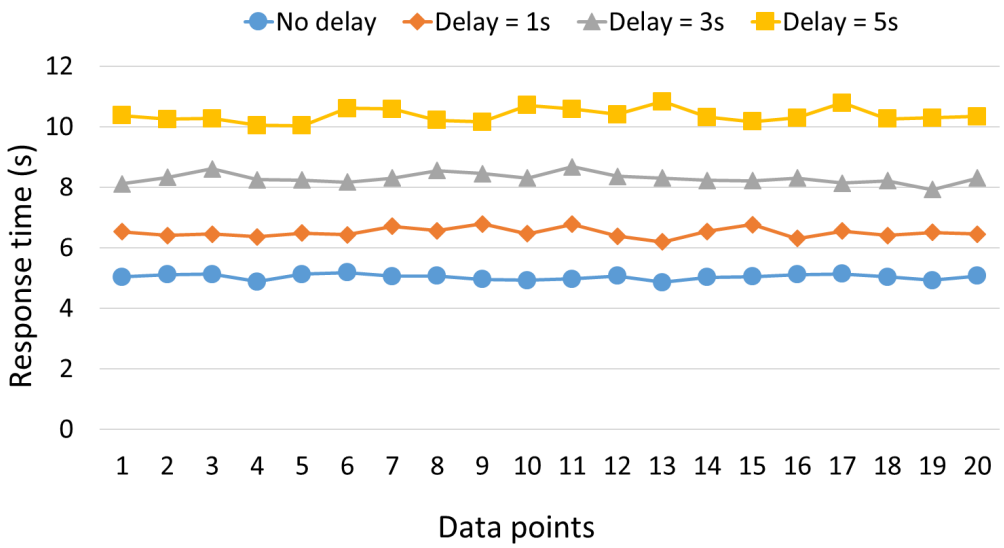

Figure 9. Time for executing a command (START, QUERY, STOP) with/ without adding delays. The CPU temperature is around 75 °C.

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 [28]) 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 trace records 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. Each I/O trace record consists of a flag (e.g., 0 indicates a read operation while 1 indicates a write operation), starting logical page address for this I/O, length of consecutive pages being read/written in this I/O. A snapshot of the trace records is shown in Figure 10, and each

record is formatted as $< flag, startingaddress, length >$. All the trace files—each stores the I/O trace records captured for each software/malware sample in our experiment—are publicly available in [29]. We used kNN to process the I/O traces. Our training set includes I/O traces of 80 malware samples and 30 benign software samples, and our test set includes I/O traces of 16 malware samples and 6 benign software samples. For each software/malware sample, we selected the first 30 I/O trace records from the corresponding trace file, and used this sequence of 30 I/Os as the feature of each sample. The dynamic time warping algorithm was used to calculate the distance between the two sequences (Section 4.1.2).

```
0  3,040  221
1  2,752  509
0  3,261  1
1  3,261  1
1  96,392  37
0  96,429  1
1  96,429  1
1  96,432  498
0  96,930  1
1  96,930  1
0  3,264  130
```

Figure 10. A snapshot of the captured I/O trace records.

To detect the malware in real-time, the detection should be as fast as possible. A small k with a good accuracy could achieve a good tradeoff for our application scenario. When choosing k as 1, we can achieve some good accuracy and the corresponding detection results, as shown in Table 1. The detection accuracy is 91%. This confirms the effectiveness of our FTL-based malware detector. In addition, 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. The reason is that, we have deliberately chosen those benign software samples which exhibit some similar patterns to the collected malware in the experiments. However, in practice, most of the benign software may not exhibit such similar patterns.

Table 1. The detection results using real-world malware samples.

Accuracy	False Negative Rate	False Positive Rate
91%	0%	33%

6.2. Experimental Results for Data Repairer

Evaluating the overhead for data repairing. To support repairing after malware is detected, mobiDOM needs to back up the mappings periodically. We, therefore, assess the time needed for backing up the mappings at the end of each epoch by varying the size of the external storage, i.e., 1, 5, 10, 50, and 100 M. The results are shown in Table 2. We can observe that: (1) The time for backing up the mappings is almost constant, regardless of the size of the storage. This is because the flash controller usually maintains a mapping table of fixed size based on the storage capacity, regardless of the size of the data stored. (2) The time needed for backing up the mappings is small, as the size of each mapping is much smaller compared to that of the data. To repair the data, mobiDOM only needs to retrieve the latest version of the mappings and to write it back. We, therefore, also assess the time needed for repairing by varying the size of the external storage, i.e., 1, 5, 10, 50, and 100 M. The results are shown in Table 3. We can observe that the time needed for repairing the external storage is small and does not grow with the size of the data, as mobiDOM only needs to restore the mapping table, which is small and fixed in size.

Table 2. The time needed for backing up the mappings.

size (MB)	1	5	10	50	100
time (s)	0.520828	0.527504	0.530729	0.531992	0.535536

Table 3. The time needed for restoring the external storage.

size (MB)	1	5	10	50	100
time (s)	0.503691	0.510664	0.519427	0.519476	0.520417

Assessing the recovery rate. We also assess whether mobiDOM can successfully recover the data consisting of different types of files. We have collected 50 sample files, which cover 5 categories and 20 file types, as shown in Table 4. The experimental results show that mobiDOM can successfully recover all the sample files, with a recovery rate of 100%.

Table 4. The file types tested for data recovery.

Category	File Type
text files	txt, ppt, pdf, rtf
image files	jpg, png, psd, tiff
video files	3gp, flv, mkv, mp4
audio files	mp3, ogg, wav, wma
others	zip, json, html, csv

7. Related Work

Malware detection. Aafer et al. [3] proposed to detect Android malware by relying on critical API calls, their package level information, and their parameters. Zhang et al. [4] proposed 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 [5–8] or analyze cryptographic primitives [6,8,30]. For example, Unveil [5] generates an artificial user environment and monitors desktop lockers, file access patterns and I/O data entropy; CryptoDrop [7] 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. On the contrary, mobiDOM specifically targets high-privileged malware that can compromise the operating system of the victim device.

Secure data recovery in mobile devices. Guan et al. [9] proposed Bolt, which can allow the system to be restored after bare-metal malware analysis. However, Bolt runs the malware in a controlled manner (for the analysis purpose), and the system has a clear knowledge of when the malware starts to function. Therefore, Bolt can simply disable garbage collection of the flash storage medium so that none of the original data stored in the flash will be deleted, which enables restoration of the original data later. Different from Bolt, mobiDOM aims to defend against malware, which is uncontrollable, and the system has no knowledge of when the malware comes. In addition, the use of TrustZone in Bolt is to enable restoration of memory state of the system; however, the use of TrustZone in mobiDOM is to ensure the trusted execution of the user-level manager of the malware detector (note that mobiDOM does not target restoration of the memory state as it is not designed for bare-metal malware analysis).

FlashGuard [31], MimosaFTL [32], SSD-Insider [33,34], Amoeba [35] aimed to combat ransomware by enabling data recovery after the ransomware attacks. The major differences between them and mobiDOM are: First, mobiDOM targets the general data corruption malware and is not specific for ransomware. Second, mobiDOM is more practical by allowing the FTL to securely communicate with the user-level app, which is necessary for further actions after malware is detected. mobiDOM makes it 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 other frameworks [31–35] do not consider interacting with user apps.

8. Conclusions

In this work, we have designed mobiDOM, a novel framework that can combat the strong OS-level malware by leveraging the existing hardware features of mainstream mobile computing devices. mobiDOM consists two major components, namely, a malware detector and a data recovery component. The malware detector is incorporated into the flash translation layer (FTL), which is isolated from the OS by the flash storage hardware. In this way, the detector can function correctly even if the OS is compromised by the malware. The data recovery component will protect the data, preventing them from being removed by the OS-level malware, by utilizing the out-of-place-update feature of the flash storage as well as modifying the existing garbage collector in the FTL. The data recovery component will work with the malware detector to enable the recovery of the external storage to a good historical state. Security analysis and experimental evaluation justify both the security and the effectiveness of our design.

There are some potential issues that should be further addressed next. First, the framework can only enable the restoration of the storage to a good historical state. However, data are valuable and it is necessary to restore the storage to the exact point where the storage is in a good state right before the malware attack. Localizing this exact point is challenging due to the delay of the malware detection and the mixing of the benign I/Os and the malware I/Os in the "grey period". In addition, the user may not have the necessary expertise to extract the raw data from the flash memory for data recovery, and a user-friendly tool that runs at the user level can bridge this gap.

Author Contributions: Conceptualization, Chen, N. and Chen, B.; methodology, Chen, N. and Chen, B.; software, Chen, N.; validation, Chen, N. and Chen, B.; formal analysis, Chen, N. and Chen, B; investigation, Chen, N. and Chen, B.; writing–original draft preparation, Chen, N. and Chen, B.; writing–review and editing, Chen, N. and Chen, B.; project administration, Chen, B.; funding acquisition, Chen, B. All authors have read and agreed to the published version of the manuscript.

Funding: This research was funded by National Science Foundation under grant number 1938130-CNS, 1928349-CNS, and 2043022-DGE.

Institutional Review Board Statement: Not applicable

Informed Consent Statement: Not applicable

Data Availability Statement: Publicly available datasets were analyzed in this study. This data can be found here: https://snp.cs.mtu.edu/research/drm2/MITON-V0.2.zip

Acknowledgments: We would like to thank Wen Xie for his contribution in the early stages of the work.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Statista. Development of New Android Malware Worldwide from June 2016 to March 2020. 2020. Available online: https://www.statista.com/statistics/680705/global-android-malware-volume/ (accessed on 20 May 2022).
2. Subedi, K.P.; Budhathoki, D.R.; Chen, B.; Dasgupta, D. RDS3: Ransomware Defense Strategy by Using Stealthily Spare Space. In Proceedings of the 2017 IEEE Symposium Series on Computational Intelligence (SSCI), Honolulu, HI, USA, 27 November–1 December 2017.

3. Aafer, Y.; Du, W.; Yin, H. Droidapiminer: Mining api-level features for robust malware detection in android. In Proceedings of the International Conference on Security and Privacy in Communication Systems, Sydney, NSW, Australia, 25–28 September 2013; Springer: Cham, Switzerland, 2013.
4. Zhang, M.; Duan, Y.; Yin, H.; Zhao, Z. Semantics-aware android malware classification using weighted contextual api dependency graphs. In Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, 3–7 November 2014.
5. Kharraz, A.; Arshad, S.; Mulliner, C.; Robertson, W.; Kirda, E. UNVEIL: A Large-Scale, Automated Approach to Detecting Ransomware. In Proceedings of the 25th USENIX Security Symposium (USENIX Security 16), Austin, TX, USA, 10–12 August 2016; pp. 757–772.
6. Kharraz, A.; Robertson, W.; Balzarotti, D.; Bilge, L.; Kirda, E. Cutting the gordian knot: A look under the hood of ransomware attacks. In Proceedings of the International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment, Milan, Italy, 9–10 July 2015; Springer: Cham, Switzerland, 2015; pp. 3–24.
7. Scaife, N.; Carter, H.; Traynor, P.; Butler, K.R. Cryptolock (and drop it): Stopping ransomware attacks on user data. In Proceedings of the 2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS), Nara, Japan, 27–30 June 2016; pp. 303–312.
8. Continella, A.; Guagnelli, A.; Zingaro, G.; De Pasquale, G.; Barenghi, A.; Zanero, S.; Maggi, F. ShieldFS: A self-healing, ransomware-aware filesystem. In Proceedings of the 32nd Annual Conference on Computer Security Applications, Angeles, CA, USA, 5–8 December 2016; ACM: New York, NY, USA, 2016; pp. 336–347.
9. Guan, L.; Jia, S.; Chen, B.; Zhang, F.; Luo, B.; Lin, J.; Liu, P.; Xing, X.; Xia, L. Supporting Transparent Snapshot for Bare-metal Malware Analysis on Mobile Devices. In Proceedings of the 33rd Annual Computer Security Applications Conference, Orlando, FL, USA, 4–7 December 2017; ACM: New York, NY, USA, 2017; pp. 339–349.
10. Chen, N.; Xie, W.; Chen, B. Combating the OS-Level Malware in Mobile Devices by Leveraging Isolation and Steganography. In Proceedings of the Applied Cryptography and Network Security Workshops, Kanagawa, Japan, 21–24 June 2021.
11. Open Portable Trusted Execution Environment. Available online: https://www.op-tee.org/ (accessed on 20 May 2022).
12. Xie, W.; Chen, N.; Chen, B. Poster: Incorporating Malware Detection into Flash Translation Layer. In Proceedings of the 2020 IEEE Symposium on Security and Privacy Poster Session, Virtual, 18–20 May 2020.
13. Peterson, L.E. K-nearest neighbor. *Scholarpedia* **2009**, *4*, 1883.
14. Müller, M. Dynamic time warping. *Inf. Retr. Music Motion* **2007**, 69–84.
15. Remove(3)—Linux Man Page. Available online: https://linux.die.net/man/3/remove (accessed on 20 May 2022).
16. Chang, L.P.; Kuo, T.W.; Lo, S.W. Real-time garbage collection for flash-memory storage systems of real-time embedded systems. *ACM Trans. Embed. Comput. Syst. (TECS)* **2004**, *3*, 837–863.
17. Bux, W.; Iliadis, I. Performance of greedy garbage collection in flash-based solid-state drives. *Perform. Eval.* **2010**, *67*, 1172–1186.
18. Bukasa, S.K.; Lashermes, R.; Le Bouder, H.; Lanet, J.L.; Legay, A. How TrustZone could be bypassed: Side-channel attacks on a modern system-on-chip. In Proceedings of the IFIP International Conference on Information Security Theory and Practice, Heraklion, Greece, 28–29 September 2017; Springer: Cham, Switzerland, 2017, pp. 93–109.
19. Ryan, K. Hardware-backed heist: Extracting ECDSA keys from Qualcomm's TrustZone. In Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, London, UK, 11–15 November 2019; pp. 181–194.
20. Tang, A.; Sethumadhavan, S.; Stolfo, S. CLKSCREW: Exposing the perils of security-oblivious energy management. In Proceedings of the 26th USENIX Security Symposium (USENIX Security 17), Vancouver, BC, Canada, 16–18 August 2017; pp. 1057–1074.
21. Qiu, P.; Wang, D.; Lyu, Y.; Qu, G. 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, London, UK, 11–15 November 2019; pp. 195–209.
22. Wan, S.; Sun, M.; Sun, K.; Zhang, N.; He, X. RusTEE: Developing Memory-Safe ARM TrustZone Applications. In Proceedings of the Annual Computer Security Applications Conference, Austin, TX, USA, 7–11 December 2020; pp. 442–453.
23. Zhu, M.; Gupta, S. To prune, or not to prune: exploring the efficacy of pruning for model compression. *arXiv* **2017**, arXiv:1710.01878.
24. Raspberry Pi 3 Model B. Available online: https://www.raspberrypi.org/products/raspberry-pi-3-model-b/ (accessed on 20 May 2022).
25. Mantech. LPC-H3131. 2017. Available online: https://www.olimex.com/Products/ARM/NXP/LPC-H3131/ (accessed on 17 May 2019).
26. Tankasala, D.; Chen, N.; Chen, B. A Step-by-step Guideline for Creating A Testbed for Flash Memory Research via LPC-H3131 and OpenNFM: https://snp.cs.mtu.edu/research/common/flash-memory-testbed.pdf (accessed on 20 May 2022).
27. Code, G. OpenNFM. 2011. Available online: https://code.google.com/p/opennfm/ (accessed on 17 May 2019).
28. VirusTotal. Available online: https://www.virustotal.com/ (accessed on 17 May 2019).
29. Malware I/O Traces On Nand flash (MITON) V0.2. Available online: https://snp.cs.mtu.edu/research/drm2/MITON-V0.2.zip (accessed on 20 May 2022).
30. Kolodenker, E.; Koch, W.; Stringhini, G.; Egele, M. PayBreak: Defense against cryptographic ransomware. In Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security, Abu Dhabi, United Arab Emirates, 2–6 April 2017; ACM: New York, NY, USA, 2017; pp. 599–611.

31. Huang, J.; Xu, J.; Xing, X.; Liu, P.; Qureshi, M.K. FlashGuard: Leveraging Intrinsic Flash Properties to Defend Against Encryption Ransomware. In Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, Dallas, TX, USA, 30 October–3 November 2017; ACM: New York, NY, USA, 2017; pp. 2231–2244.
32. Wang, P.; Jia, S.; Chen, B.; Xia, L.; Liu, P. MimosaFTL: Adding Secure and Practical Ransomware Defense Strategy to Flash Translation Layer. In Proceedings of the Ninth ACM Conference on Data and Application Security and Privacy, CODASPY '19, Dallas, TX, USA, 25–27 March 2019; ACM: New York, NY, USA, 2019.
33. Baek, S.; Jung, Y.; Mohaisen, A.; Lee, S.; Nyang, D. SSD-insider: Internal defense of solid-state drive against ransomware with perfect data recovery. In Proceedings of the 38th IEEE International Conference on Distributed Computing Systems, ICDCS 2018, Vienna, Austria, 2–5 July 2018.
34. Baek, S.; Jung, Y.; Mohaisen, A.; Lee, S.; Nyang, D. SSD-assisted Ransomware Detection and Data Recovery Techniques. *IEEE Trans. Comput.* 2020, 70, 1762 - 1776.
35. Min, D.; Park, D.; Ahn, J.; Walker, R.; Lee, J.; Park, S.; Kim, Y. Amoeba: an autonomous backup and recovery SSD for ransomware attack defense. *IEEE Comput. Archit. Lett.* **2018**, *17*, 245–248.