Enabling Accurate Data Recovery for Mobile Devices against Malware Attacks

Wen Xie, Niusen Chen, and Bo Chen*

Department of Computer Science, Michigan Technological University, Michigan, United States
bchen@mtu.edu

Abstract. Mobile computing devices today suffer from various malware attacks. After the malware attack, it is challenging to restore the device's data back to the exact state right before the attack happens. This challenge would be exacerbated if the malware can compromise the OS of the victim device, obtaining the root privilege. In this work, we aim to design a novel data recovery framework for mobile computing devices, which can ensure recoverability of user data at the corruption point against the strong OS-level malware. By leveraging the version control capability of the cloud server and the hardware features of the local mobile device, we have successfully built MobiDR, the first system which can ensure restoration of data at the corruption point against the malware attacks. Our security analysis and experimental evaluation on the real-world implementation have justified the security and the practicality of MobiDR.

Keywords: mobile device, data recovery, OS-level malware, corruption point, FTL, TrustZone, version control

1 Introduction

Mainstream mobile computing devices (e.g., smart phones, tablets, etc.) have been suffering from various malware attacks [7]. For example, ransomware encrypts the data of a victim device and asks for ransom; trojans first steal data from a victim device, send the data to the remote controller, and corrupt the data locally. Especially, there is one type of strong malware [14] which can compromise the OS, obtaining the root privilege. This type of OS-level malware is difficult to combat due to its high system privilege [14]. Data are extremely valuable for both organizations and individuals. Therefore, after a mobile device is attacked by the OS-level malware and the stored data are corrupted, it is of significant importance to ensure that the valuable data can be restored to the exact state right before the malware corruption (**data recovery guarantee**). We define the point of time right before the malware starts to corrupt the data as the "corruption point", and the data recovery guarantee requires restoring the data at the corruption point after malware attacks.

* Corresponding author.

To enable data recovery, existing works either 1) purely rely on a remote version control server [19, 11, 18], or 2) purely rely on the local device [28, 29, 8, 9, 21, 23]. Simply relying on the remote version control server cannot achieve the data recovery guarantee, as the OS-level malware may compromise the most recent data changes (i.e., *delta*) in the device which have not[1] been committed remotely and, the remote server can only allow restoring the historical state of the data rather than the exact state at the corruption point. Simply relying on the local storage cannot achieve the data recovery guarantee against the OS-level malware either, because: First, FlashGuard [21] and TIMESSD [29] retain historical data in the storage hardware for data recovery. This is essentially equal to maintaining a local version control system but, due to the limited local storage capacity, the historical data can only be retained for a short term (e.g., 20 days in FlashGuard). This implies that the data which are not retained any more may become irrecoverable if compromised by the malware. Second, MimosaFTL [28], SSD-Insider [8], and Amoeba [23] incorporate malware detection to avoid retaining too much unnecessary historical data, but the malware detection may suffer from false negatives and the data corrupted by the undetected malware may be lost. SSD-Insider++ [9] tries to compensate the false negatives, but their strategy is specific for the ransomware and, their lazy detection algorithm still suffers from potential false negatives.

In this work, we aim to achieve the data recovery guarantee against the OS-level malware. Our key idea is to build a secure version control system *virtually* across the mobile device and the version control server in an adversarial setting (Figure 1), such that the most recent delta data are correctly maintained in the mobile device and the historical delta data are correctly stored in the cloud server. In this manner, any version of data is recoverable in the mobile device hence the version of data at the corruption point is always recoverable. A salient advantage of our design is that it does not rely on any malware detection mechanisms and hence does not suffer from false negatives and, meanwhile, it does not suffer from the storage capacity as the cloud storage can be easily scaled up. Towards the aforementioned goal, the first step of our design is to ensure that the OS-level malware cannot corrupt any newly generated delta data. Mobile devices today usually use flash memory as external storage and, a flash storage medium typically exhibits two salient hardware features: 1) performing out-of-place update internally, and 2) introducing a flash translation layer (FTL) to transparently handle the flash memory hardware. Therefore, we can simply hide the delta data in the flash memory [20, 21]: due to the physical isolation, the malware cannot physically "damage" the delta data stored in the flash memory even if it can compromise the OS; additionally, as the flash storage performs out-of-place update, overwriting operations performed by the malware at the OS level can only invalidate rather than delete the delta data stored in the flash memory. Besides, our design needs to address extra challenges:

[1] Typically, delta data are committed to the remote server periodically rather than continuously, to reduce bandwidth/energy consumption imposed on the low-power mobile computing devices.

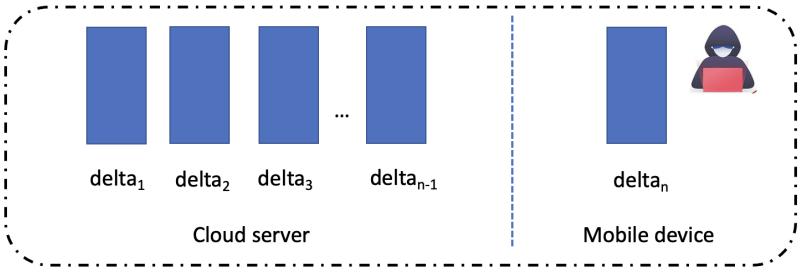


Fig. 1. A virtual version control system across the cloud server and the mobile device. We focus on defending against attackers in the mobile device side. Handling attackers in the cloud server side has been explored extensively before [19, 11, 17, 18, 31, 12, 26].

First, the malware may first overwrite the user data at the OS level, invalidating them in the flash memory, and then fill the entire disk (on top of the block device) to force the garbage collection in the flash memory to reclaim those flash blocks storing invalid data. To address this challenge, we periodically invoke a backup process which commits the new delta data to the cloud server and, before any new delta data are committed remotely, we will temporarily "freeze" the garbage collection over them. In other words, once the new delta data are committed remotely, the garbage collection on them can run normally.

Second, the OS-level malware may disturb the backup process, so that the delta data may not be securely extracted from the flash memory and correctly committed to the cloud server. To facilitate the backup process, we need a backup app which runs in the application layer, securely extracting delta data and committing them to the cloud server. Two issues need to be tackled:

1) How can we prevent the backup app from being compromised by the OS-level malware? Mobile devices today are broadly equipped with Arm processors, which provide a hardware-level security feature TrustZone. TrustZone can allow creating a trusted execution environment (i.e., a secure world) and, any code running in this environment cannot be compromised by the adversary which can compromise the OS. We therefore move critical components of the backup app into the secure world to avoid being compromised by the malware.

2) How can the backup app securely extract delta data from the flash memory? The backup app runs at the application layer and does not have access to the raw data in the flash memory. We therefore modify the FTL, so that upon the backup process, it can work with the backup app, extracting the raw flash memory data and sending them to the backup app. To prevent the malware from disturbing the extraction process, we incorporate a backup mode into the FTL and, if the mode is activated, the FTL will be exclusively working for the backup process. To activate the backup mode securely, authentication is performed based on the secret known by the backup app and the FTL. To prevent the malware from corrupting delta data sent from the FTL to the backup app, the FTL will compute cryptographic tags for the delta data using a secret key, and the backup

app will verify the delta data before committing them remotely. To prevent the malware from replaying old delta data, the version number should be embedded into each tag. Note that the FTL stays isolated from the OS, therefore the malware cannot compromise the tag computing process as well as the secret key.
Contributions. We summarize our contributions below:

– We have designed MobiDR, the first data recovery system for mobile computing devices, that can ensure recoverability of data at the corruption point against malware attacks. We consider the strong malware which is allowed to compromise the OS of the victim device.
– We have built two user-level apps, DRBack and DRecover, which make the proposed design usable by regular mobile users in the user space. The apps work together with our modified FTL (DRFTL) to enable secure data backup (periodically) and data recovery (upon failures).
– We have analyzed the security of MobiDR. In addition, we have implemented a real-world prototype of MobiDR using a few embedded boards and a remote version control server, and assessed the performance of MobiDR.

2 Background

Flash memory. Flash memory especially NAND flash is widely used as the mass storage for mobile devices today. For instance, main-stream smartphones and tablets use eMMC and UFS cards; smart IoT devices use microSD cards. Flash memory is typically organized into blocks, each of which is further divided into pages. The number of pages in a flash block varies from 32 to 128, and the page size varies from 512, 2048, to 4096 bits. Each page usually contains a small spare out-of-band (OOB) area, used for storing metadata like error correcting code. Flash memory typically supports three operations: read, program, and erase. The read/program operation is performed on the basis of pages, while the erase operation is performed on the basis of blocks. Different from conventional mechanical drives, flash memory exhibits some unique characteristics. First, it follows an erase-then-write design. This means, re-programming a flash page usually requires first erasing it. However, since the erase operation can only be performed on blocks, re-programming a few pages would be expensive. Therefore, flash memory uses an *out-of-place update* strategy for small writes. Second, each block can be programmed/erased for a limited number of times, and a block would be worn out if it is programmed/erased too often. Therefore, wear leveling is usually integrated to distribute writes/erasures across the flash evenly.
Flash translation layer (FTL). Flash memory exhibits completely different nature compared to HDDs (hard disk drive). To be compatible with traditional block file systems (e.g., EXT4, FAT32) built for HDDs, a flash-based storage device is usually used as a block device. This is achieved by introducing an extra firmware layer, namely, the flash translation layer (FTL) to transparently handle unique characteristics of flash memory, exposing a block access interface. The FTL stays isolated from the OS, implementing a few unique functions including address translation, garbage collection, wear leveling, and bad block

management. Address translation maintains the address mappings between the addresses (i.e., the Logical Block Addresses or *LBAs*) accessible to upper layers and the flash memory addresses (i.e., the Physical Block Addresses or *PBAs*). Garbage collection periodically reclaims the flash memory space occupied by obsolete data which have been invalidated by the FTL after the out-of-place update is performed. Wear leveling periodically swaps blocks so that the programmings/erasures performed over the entire flash blocks can be even out. Bad block management handles those blocks which have been worn out.

TrustZone. ARM TrustZone is a main-stream trusted execution environment (TEE) implementation for mobile devices. It is a hardware-based technology which provides security extension to ARM processors[2]. TrustZone separates two worlds, a secure world and a normal world. The two worlds have isolated memory space and different privilege level to peripherals. Applications running in the normal world cannot access memory space of the secure world, while applications running in the secure world can access memory space of the normal world in certain conditions. The processor can only run in one world at a certain time. A special *Non-secure* (NS) bit determines in which world the processor is currently running. A privileged instruction *Secure Monitor Call (SMC)* switches the processor between the normal and the secure world.

3 System and Adversarial Model

System model. Our system mainly consists of two entities (Figure 2): 1) a mobile computing device; and 2) a remote cloud server. The mobile device is equipped with a flash-based block device as external storage (e.g., an eMMC card, a microSD card, or a UFS card) and Arm processors with TrustZone enabled. The flash memory is transparently managed by the *FTL*, exposing a block access interface. The TrustZone can separate two worlds in the mobile device, a normal world running untrusted applications, and a *secure world* running *trusted applications* (TA). The cloud server runs a version control system and interacts with the mobile device. As the server is running as a cloud instance, its computational resources (i.e., computing power, data storage) can be easily scaling up and down according to the need.

Adversarial model. In the mobile device, we mainly consider an adversary (Figure 2) which *performs data corruption attacks*, i.e., data corruption malware. This can be ransomware which encrypts a victim device's data and asks for ransom. This can also be a piece of trojan or backdoor malware that first steals user data and then damages them locally in the victim device. We consider the strong OS-level malware [14] which can compromise the regular OS running in the TrustZone normal world and corrupt any data visible to the OS. Here "data" especially refers to the information having been committed to the external storage rather than those staying in the memory and not yet been committed.

The cloud server is assumed to correctly store and maintain the versioning data, and how to ensure integrity of the versioning data outsourced to an

[2] TrustZone has been broadly supported since ARMv7.

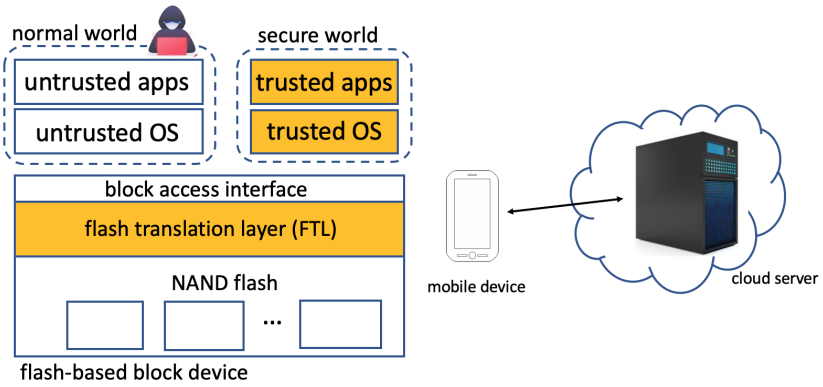


Fig. 2. Our system model. The part on the left is the architecture of the mobile device, in which the components in yellow color are isolated from the untrusted OS.

untrusted remote server has been explored extensively in prior works [19, 11, 17]. Other assumptions include: 1) TrustZone is secure, and the malware cannot compromise the secure world, including the trusted applications and data in it. This is a reasonable assumption in the domain of TrustZone technologies [20]. Although a few security leaks [30, 24] have been found in TrustZone, hardening TrustZone has been explored extensively in the literature [27] and is not our focus. 2) The malware is not able to hack into the FTL. This is also reasonable as the FTL is isolated from the OS (Figure 2) and there are no known attacks which can bypass the isolation utilizing the limited block access interface exposed by the flash device. 3) The communication channel between the mobile device and the cloud server is assumed to be protected by TLS/SSL. 4) The malware will not perform arbitrary behaviors like blocking user I/Os or conducting DoS attacks which would be easily noticed by the device's owner.

4 MobiDR

4.1 Design Rationale

To achieve the data recovery guarantee against the OS-level malware, MobiDR relies on the versioning data in the cloud server as well as the most recent delta stored in the local device. Periodically, MobiDR securely commits changes of data (i.e. delta) in the local device to the cloud server (*the backup phase*). After a malware attack, MobiDR will retrieve the versioning data from the server, applying the most recent delta (stored locally) to restore the data at the corruption point (*the recovery phase*).

The backup phase. The backup phase happens *periodically*. After each successful backup, the FTL will monitor write requests from the OS and, if a new flash page is written, it will push the corresponding PBA into a stack and the

garbage collection on those new flash pages should be frozen[3] temporarily even if they are invalidated. Meanwhile, the FTL will monitor write requests on some reserved LBAs. Note that the LBAs are accessible to the OS, e.g., if a disk sector is 512-byte, and a flash page is 2KB, every 4 sector addresses can be converted to one LBA. If a secret write sequence on the reserved LBAs is observed by the FTL, a new backup phase has been invoked by the backup app and the FTL will enter the backup mode. In the backup mode, the backup app will work with the FTL to extract the most recent delta data securely from the flash memory and to correctly commit them to the cloud server. The backup app will issue read requests (i.e., by writing the reserved LBAs) and, upon receiving a read request, the FTL will: 1) pop a PBA from the stack, read the data stored in the corresponding flash page, and identify the corresponding LBA; and 2) compute a cryptographic tag over a concatenation of the data, the LBA, the current version number as well as the sequence number (during each backup process, the sequence number starts from 0, and is increased by 1 after each read request), using a secret key; and 3) return the data, the LBA, and the tag for each read request. Upon receiving a response from the FTL, the backup app will verify the integrity of the data using the tag and the secret key. Once the stack is exhausted by the FTL, the backup app will verify and commit all the delta data together with their tags to the cloud server, and quit the backup mode. Note that critical components of the backup app should be run in the TrustZone secure world to avoid being compromised by the OS-level malware.

The recovery phase. Once a mobile device suffers from a malware attack and the stored data are corrupted, a data recovery phase will be activated to restore the data back to the *corruption point*. The malware is assumed to have been detected [14] at some point of time (i.e., the *detection point*) and eliminated[4] from the victim device and, therefore, the recovery app can be run in the normal world. As the backup phase is activated periodically, the device should have conducted a successful backup process (i.e., the *most recent backup point*) before the malware is detected. The corruption point should be located either between the most recent backup point and the detection point (if the malware detection is effective and can detect the malware immediately) or before the most recent backup point (if the malware detection suffers from false negatives). The recovery app will restore the data at the corruption point by: 1) retrieving versioning data from the remote server (correctness of the data is verified via the cryptographic tags), and 2) extracting the most recent delta preserved locally in the flash memory, and 3) approaching the corruption point by interacting with the user following a binary searching manner (details elaborated in DRecover of Sec. 4.2).

[3] An extreme case is that the device is almost filled and there are no unused blocks. In this case, if there is a flash block which stores invalid data that have not been backed up yet, MobiDR will back up those data immediately and garbage collection can be immediately performed on this block.

[4] If the malware is impossible to be eliminated, we can unplug the flash storage medium from the victim device and plug it into a clean device for the recovery phase.

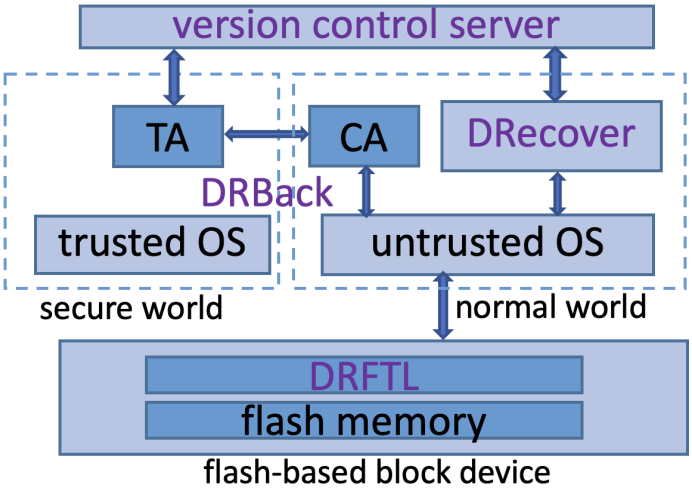


Fig. 3. The design overview of MobiDR.

4.2 Design Details

Overview. The overview of MobiDR is shown in Figure 3. The version control server runs a version control system which allows storing and retrieving versioning data. The DRFTL is a special flash translation layer tuned for MobiDR design, which can transparently manage the raw NAND flash and work with the user-level apps for data backup and recovery. The DRBack app consists of a client application (i.e. *CA*) which runs in the normal world (based on a rich untrusted OS) and a trusted application (i.e. *TA*) which runs in the secure world of TrustZone. The *backup phase* is conducted periodically by the DRBack app, which communicates with both the DRFTL via the OS (for extracting delta data from the flash memory) and the remote version control server (for storing the delta data remotely). The *recovery phase* is conducted by the DRecover app. The DRecover app communicates with the remote version control server (for retrieving necessary versioning data) and the DRFTL (for extracting the most recent delta preserved in the flash memory, reconstructing the data at the corruption point as well as placing the data back to the flash memory). In the following, we elaborate the design detail of each major component in MobiDR.

The version control server. The cloud server runs a version control system [11, 19] which allows the client to commit a new version of the data (e.g., via commit), or to retrieve an arbitrary historical version (e.g., via checkout). In MobiDR, the data committed during each backup phase are the most recent delta currently and the corresponding cryptographic tags. The data retrieved during the recovery phase are the collection of deltas (and their corresponding tags) from the initial version up to an arbitrary version. Each delta is a collection of raw data newly stored to the flash memory pages.

DRFTL. The DRFTL will keep track of delta data (not yet committed remotely), protecting them from being deleted by the malware. It will also collaborate with the DRBack to correctly extract and commit the delta data, and collaborate with the DRecover to restore the data to the corruption point.

To protect the delta data in the flash memory, we need to understand the delta a bit. Here the delta means the changes of data in the flash memory. In the user space, there are typically three operations: read, write, and delete. The read operation usually does not generate delta. The delete operation is typically handled by the OS as follows: the OS will mark the corresponding disk space as invalid by updating its metadata which may cause an overwrite operation in the flash memory. The write operation in the user space may either cause a new write or an overwrite in the flash memory. For a new write, the new content will be placed to a new PBA (corresponding to a physical flash page) in the flash memory; for an overwrite, the obsolete content in the old PBA will be invalidated, and the new content will be placed to a new PBA due to the out-of-place update. Therefore, our observation is that, the new content generated since the last backup process is always stored in the new PBAs and, therefore, we can simply keep track of all the new PBAs in the flash memory, following the order they are written. In addition, each flash page has an OOB area which typically records its corresponding LBA. Therefore, the content in a PBA contains all the information[5] needed for the recovery. Note that the malware cannot compromise the delta as the flash memory performs the out-of-place update, and only the garbage collection in the FTL can remove data.

To keep track of new PBAs, the DRFTL maintains a stack in the internal RAM of the flash device. To avoid data loss due to unexpected instances like power loss, we should periodically commit the data in the stack to the flash and, once the backup phase is finished, the data associated with the stack can be cleared from both the RAM and the flash. Once a backup phase is invoked, the DRFTL will pop a PBA from the stack, read the content from the corresponding flash page, and return it to the DRBack. The aforementioned step will be terminated when all the delta data are extracted (i.e., the stack is empty).

In order to differentiate the backup/recovery phase with the normal use, we define a backup and a recovery mode for the FTL, respectively. In the backup mode, the FTL will exclusively work with the DRBack for extracting and committing the delta and, in the recovery mode, the FTL will exclusively work with the DRecover to restore the data to the corruption point. Since only the DRBack knows when to enter the backup mode, it should inform the DRFTL once it launches the backup phase. This can be achieved by reserving some LBAs, and the DRBack will perform writes on those LBAs with some secret sequence known to the DRFTL. To avoid replay attacks, we can concatenate the sequence with an index (increased by one upon a new backup phase) and encrypt it using the secret key shared between the DRBack and the DRFTL. Similarly, the DRecover

[5] During recovery, we can simply place the content back to the LBA in the flash memory, since where the content will be physically located is not important.

will inform the DRFTL once it launches the recovery phase with another secret write sequence on the reserved LBAs.

To prevent reclaiming flash blocks which store the delta data that have not been committed remotely, the garbage collection on the delta data will be disabled temporarily, but will be resumed as soon as a following backup is finished (in which the delta data are committed to the remote server).

DRBack. The DRBack will work with the DRFTL to extract the most recent delta data from the flash memory, and send them to the cloud server after having verified their correctness. Note that the DRBack contains both a trusted application (TA) running in the TrustZone secure world (mainly responsible for verifying and committing the delta), and an untrusted client application (CA) running the normal world (used as a proxy for the TA to communicate with the DRFTL to extract the delta data).

Periodically, the TA of the DRBack will issue a secret write sequence (using CA as a proxy) to the reserved LBAs, changing the DRFTL to a backup mode. In the backup mode, the TA continuously reads a special LBA until having extracting all the new delta data since the last backup process. When the DRFTL receives a read request from the TA, it will read a flash page (the PBAs are kept in the stack), and compute a cryptographic tag for the data of the page. To defend against the replay attack, the current version number, the page sequence number, the LBA and the data of the page are combined together when computing the tag. The DRFTL will return the data of the page in the first read; the LBA, the current version number, the page sequence number in the current version, as well as the tag will be combined and returned in the second read. Therefore, to extract the delta data stored at a single page, the TA needs to perform two read operations. After all the new delta data are read, the DRFTL will inform the TA (this can be achieved by responding with some special content to the read request issued by the TA). The TA will verify the delta data and, if they are correct, the TA will commit them to the cloud server. It will then send a confirmation (together with a cryptographic tag, computed over the confirmation and the version number via the secret key shared between the TA and the DRFTL) to the DRFTL and, after the DRFTL has successfully verified the confirmation, it will quit the backup mode and return to a normal state.

DRecover. The DRecover will collaborate with the DRFTL to restore the data to the corruption point. The DRecover will first issue a different secret write sequence to the reserved LBAs to change the DRFTL to the recovery mode; it will then retrieve the most recent delta data from the flash memory. The most recent delta data can be stored in another computing device or committed remotely. Next, it will retrieve the most recent version of the data from the cloud server, verify its integrity, and place them back to the flash memory. The user needs to check whether this restored data version has any corruptions or not. If it has no corruptions, the corruption point is located somewhere between the most recent backup point and the detection point (*case #1*); otherwise, the corruption point is located somewhere between the initial point and the most recent backup

point (*case #2*). After the recovery phase is finished, the DRecover will change the DRFTL back to the normal state.
Handling case #1: The DRecover will first restore the data to the most recent backup point, and then sort the most recent delta data based on the timestamps of each page in the increasing order (for simplicity, we call them the sorted delta data). The DRecover places the first half[6] of the sorted delta data back to the flash memory, and the user will check whether this restored version contains corrupted data or not. If it contains, the corruption point should be moved backwards; otherwise, the corruption point should be moved forwards. A binary searching will be continued in either half, recursively. The number of user involvement will be $O(log\ l)$, where l is the total number of pages in the sorted delta. Note that the user involvement seems to be unavoidable, as only the user knows whether his/ her data were corrupted or not.
Handling case #2: The DRecover will retrieve a historical version from the remote server which is at the middle of the initial point and the most recent backup point, and work with the DRFTL to place this version back to the flash memory. The user will check whether this restored version contains corrupted data or not. If it does, the corruption point is located at a point even earlier; otherwise, the corruption point is located at a point later. A binary searching will be continued in either half, recursively. After a target version is located, the corruption point can be further located similar to case #1. The number of user involvement will be $O(log\ n + log\ l)$, when n is the total number of historical versions stored in the remote server and l is the maximal number of pages in a delta.

5 Security Analysis and Discussion

Security Analysis. In the following, we show that MobiDR can ensure recovery of data at the corruption point against the OS-level malware.
Any newly created delta can be correctly committed to the remote server during the backup phase. The newly created data which have been written to the flash memory will not be deleted by the garbage collection of the DRFTL before they are committed to the remote server. Therefore, regardless how the malware behaves at the OS level, e.g., over-writing the user data at the block layer to invalidate them in the flash memory, writing arbitrary data to the disk sectors, the new data will stay intact in the flash memory. Note that the DRFTL is transparent to the OS, and will not be affected malware. The DRBack runs in the user space, which is separated into two parts: one (CA) is running in the normal world and acts as a proxy to communicate with the DRFTL, and the other (TA) is running in the secure world and is responsible to verify the extracted data and to commit them remotely after the verification. The malware may affect the CA, e.g., when the CA is used as a proxy to extract data from the DRFTL, the malware may corrupt the data passing through the untrusted

[6] The description here is not very exact. In practice, a few pages together may belong to the same atomic operation and cannot be separated.

OS. This corruption attack can be mitigated as the DRFTL will compute cryptographic tags for the extracted data and, the corruption will be detected by the TA which will not be affected by the malware. If the corruption is detected, the TA will require the CA to extract the data again (note that if the corruption persists, the TA should notify the user, as there is a potential DoS attack); others, the TA will send the extracted data, together with the associated tags, to the server.

The most recent delta data will not be corrupted by the malware. After the latest backup process, any new data created by the user since then, cannot be compromised by the OS-level malware once they have been written to the flash memory. This is because, the data stored at the flash memory are not accessible to the OS, and can only be removed by the garbage collection of the FTL; however, DRFTL has modified the garbage collection strategy so that any newly created data, valid or not, will not be deleted from the flash memory before they are correctly committed to the remote server.

MobiDR **can always recover data at the corruption point during the recovery phase**. During the recovery phase, the device is always in a healthy state, either because the malware has been eliminated from the victim device or because a clean device has been used for recovery. Therefore, the DRecover can run correctly in the OS. An arbitrary historical version of the data can be retrieved correctly by the DRecover from the version control server (crytographic tags are used to verify the correctness of a historical version upon retrieval). In addition, the new data changes since the most recent committed version are preserved in the flash memory, and are extractable by the DRecover. Being able to have access to any version of the historical data, as well as the most recent delta data, the DRecover is able to restore data at any point over the history, surely including the corruption point.

Discussion. In the following, we discuss a few minor issues in MobiDR.

Sharing secret keys between the DRFTL and the TA. During initialization, the device owner can generate the secret keys, and send them to the TA in the secure world; in addition, the secret keys can be passed to the FTL as follows: the FTL reserves an LBA and monitors the writes on this page; once the device owner writes the secret keys to this LBA, the FTL will read it, copy the keys to other area invisible to the OS, and clear the data in this LBA.

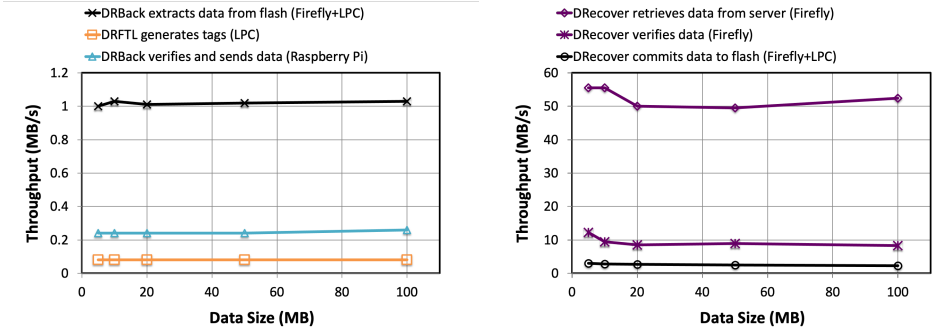
Handling device failures. If the device fails (e.g., suffering from power loss [22]) upon backup and the most recent delta has not been committed yet, the user could try mobile device forensics [10] to extract the most recent delta from the device, though there is no guarantee whether the delta can be extracted or not. If the device is lost/stolen, there is still a possibility that the latest delta would be backed up to the remote server if the "pickpocket" turns on the device and network connection is available for the device. In the worst case, the user can at least restore the data to the latest backup stored in the remote server. It seems no approach can completely address the aforementioned limitation, unless the device backs up every single operation to the remote server which is impractical.

The impact of "freezing" garbage collection. Freezing the garbage collection will not affect the system much because: 1) The garbage collection is only frozen for those data not yet been committed remotely. 2) The garbage collection on the not-yet-committed data is only frozen for a short period, e.g., if the device is backed up daily, the period is one day. 3) If the malware fills the entire storage on purpose (i.e., no unused flash blocks), the backup operation will be performed immediately, and the garbage collection will run normally after it.
Moving the entire DRBack into the TrustZone secure world. One alternative is to move the entire DRBack to the secure world to prevent DoS attacks conducted by the malware. However, accessing the external storage in the secure world is non-trivial. This may require incorporating extra software components into the secure world, including disk driver and other components along the storage path [14]. We will investigate this alternative in our future work.

6 Experimental Evaluation

We have implemented a prototype of MobiDR, which includes DRFTL, a DRBack app, a DRecover app, and a server program. DRFTL was implemented by modifying OpenNFM [15], an open-sourced flash controller framework written in C. The cryptographic tag was instantiated using HMAC-SHA1. The DRBack app, the DRecover app, and the server program were all written in C. The DRBack app consists of two major software components (both were implemented in C): one software component (CA) runs in the normal world, and the other software component (TA) runs in the secure world of ARM TrustZone. Our TA relies on the support of OP-TEE [3], an open source trusted execution environment implementing Arm TrustZone technology, which has been ported to many Arm devices and platforms.
Experimental setup. We ported the DRFTL to LPC-H3131 [2], a USB header development prototype board with ARM9 32-bit ARM926EJ-S (180Mhz), 32MB SDRAM, and 512MB NAND flash. After the DRFTL is ported, LPC-H3131 can be used as a flash-based block device via USB 2.0. Both the DRBack (CA) and the DRecover were run as an application in another electronic development board Firefly AIO-3399J [1], equipped with Six-Core ARM 64-bit processor (up to 1.8GHz) and 4GB Dual-Channel DDR3. AIO-3399J acts as the host computing device of the mobile device to perform I/Os on the flash storage provided by LPC-H3131. Note that although the processor of Firefly AIO-3399J can support TrustZone, the manufacturer of this electronic board does not offer a free support for TrustZone development. We instead measured the TA of DRBack in TrustZone secure world provided by a cheap Raspberry Pi (version 3 Model B, with Quad Core 1.2GHz Broadcom BCM2837 64bit CPU, 1GB RAM) [4], by porting OP-TEE[7] to it. The remote version control server was run by a desktop (8 core Intel Core i7-9700K CPU, 3.60 GHz, 32GB RAM), which was connected to a local area network in our lab, and the electronic boards (Firefly AIO-3399J

[7] Currently OP-TEE has not supported TLS yet, which can be implemented as "a glue layer between mbedTLS and the GP API provided [5]".



(a) The throughput of each component in the backup phase.

(b) The throughput of each component in the recovery phase

Fig. 4. Performance in the backup and the recovery phase.

and Raspberry Pi) were both connected to the same local area network. Note that none of the prior works (Sec. 7) can ensure recoverability of the data at the corruption point over time under an adversarial setting; therefore, we did not experimentally compare MobiDR with them, considering that the goal of MobiDR is different from all of them and the comparison would not be fair.

Evaluating The Backup Phase. The backup process is conducted by DRBack, together with DRFTL and the remote version control server. The DRBack needs to first extract data from the raw flash memory to the user space, by working with the DRFTL. This sub-process is taken care by the CA of DRBack, which runs in the normal world (rather than TrustZone secure world). The DRFTL computes tags for the data being extracted. As shown in Figure 4(a), the throughput for data extraction is approximately 1MB/s, which is regular for a USB 2.0 interface. The throughput for computing the tags in LPC-H3131 is around 100KB/s. This is reasonable for a low-power electronic board equipped with a 180MHz processor. In practice, we can replace the cryptographic hash function SHA1 with a more efficient non-cryptographic hash function like XXH128, which was shown 30 times faster than SHA1 [6].

After having extracting the data, the DRBack will verify integrity of the data based on the associated tags, and send them (together with tags) to the remote server if the verification is successful. This sub-process is taken care by the TA of DRBack, which runs in the TrustZone secure world. As shown in Figure 4(a), the throughput for these TA operations is approximately 250KB/s. The major computation in the TA (running in the TrustZone of Raspberry Pi) is to verify the correctness of the tag (HMAC-SHA1), which requires a similar computing workload compared to the tag generation (running in the LPC-H3131). The performance of the tag verification in Raspberry Pi is 2×-3× better than the tag generation in LPC-H3131, which makes sense since Raspberry Pi is more powerful than LPC-H3131.

Evaluating The Recovery Phase. The recovery phase is conducted by DRecover, together with DRFTL and the remote version control server. If the corruption happens after the most recent backup process, MobiDR will restore

the device by retrieving the most recent data version from the remote server, and applying the local delta up to the corruption point. If the corruption happens before the most recent backup process, it implies that the malware detection cannot detect the malware timely, or suffer from false negatives in the past. Therefore, the corruption point should be located anywhere from the initial data version to the most recent data version. For this case, we can retrieve the most recent data version, and localize the data version which is before but closest to the corruption point. The data at the corruption point can be restored by starting from the closest version, and applying the corresponding delta up to the corruption point. In the following, we assess the performance of two key steps: 1) retrieving the most recent data version from the remote server; and 2) localizing the closest data version from the most recent data version locally. As MobiDR performs data recovery by centering around the raw data in the flash memory (rather than the traditional file data), we also access whether it can recover a given file accurately or not.

Retrieving the most recent data version from the server. To retrieve the most recent data version, DRecover will retrieve all the deltas from the server, verifying them, and committing them back to the flash memory (by working with the DRFTL). The experimental results are shown in Figure 4(b). The throughput for data retrieval is around 50MB/s, which is reasonable since the communication happens in a local area network. The throughput for data verification is around 9MB/s. This is because, Firefly AIO-3399J is a high-end electronic board with performance comparable to the desktop. The throughput for data commit is around 2.5MB/s. This is reasonable for a USB 2.0 interface. Compared to the data extraction in Figure 4(a), the throughput for data commit is 2× faster. This is because, the data extraction requires extra read operations for obtaining the tags, but the data commit does not need to write the tags.

Localizing the closest data version. We have conducted an experiment in which there are 64 data versions in total, and the delta size is 2MB (i.e., each version will generate 2MB additional data compared to its immediate previous version). After the DRFTL has retrieved the most recent data version, it will localize a closest data version following a binary search manner, i.e., a new data version will be restored in the device and the user will get involved to determine how the "search" will be moved next. The results are shown in Table 1. We can observe that, if the targeted data version is 32, it will be localized with the minimal time, since the first version to be examined is version 32 based on the rule of binary search, and the total number of user involvements is 1; similarly, if the targeted data version is 16 or 48, it will take more time compared to version 32, since DRFTL needs to first examine version 32, and then examine version 16 or 48 (depending on the user feedback), and the total number of user involvement is 2. Without knowing where is the close data version, binary search would require at most log(n) user involvements, and the total number of versions needed to be examined is also bounded by log(n).

Recovery rate. To evaluate whether MobiDR can recover a given file accurately by placing the raw data back to the flash memory, we tested 100 sample files,

Closest version number	Time (s)	#User involvement
8	39.10	3
16	33.84	2
32	22.96	1
48	35.17	2
56	41.63	3

Table 1. The overhead for localizing a closest data version in DRecover, in which the most recent data version is 64, and each delta size is 2MB.

covering 5 categories and 30 file types (see Table 2), with file size varying between 1MB to 100MB. The results show that MobiDR can accurately recover all of them, which indicates a recovery rate of 100%.

category	file type
text files	txt,pdf,rtf,ppt,odp,doc
image files	jpg,webp,tiff,gif,psd
video files	flv,mkv,3gp,mp4,wmv,webm,avi,f4v
audio files	mp3,ogg,wav,flac
others	zip,bin,db,tar,img,exe,msi

Table 2. Summary of sample files used for testing the recovery rate.

7 Related Work

Continella et al. designed ShieldFS [16], a self-healing, ransomware-aware file system. ShieldFS can automatically shadow a copy whenever a file is modified and, the shadow copy can be used to recover the file corrupted by the ransomware. Subedi et al. proposed RDS3 [25], which hides backup data to an isolated storage space for data recovery. Both ShieldFS and RDS3 cannot combat the malware which can compromise the OS, as they are both deployed at the OS level.

Huang et al. proposed FlashGuard [21] to enable data recovery from ransomware attacks. FlashGuard needs to preserve all the historical versions of "possibly" attacked data locally in the flash memory to maximize probability of successful recovery. Wang et al. proposed TIMESSD [29] to enable data recovery by retaining past storage states in the local SSD. FlashGuard and TIMESSD try to support data recovery via the local version control, and both unavoidably suffer from the limited storage space in the local device. SSD-insider (Baek et al. [8]), and MimosaFTL (Wang et al. [28]) and Amoeba (Min et al. [23]) improved FlashGuard by incorporating a ransomware detection into the FTL, so that the local device does not need to preserve invalid data in the flash memory if the ransomware is not detected. This can save local storage space, but the malware detection unavoidably suffers from false negatives and, if a false negative happens, the data corrupted by the ransomware will become irrecoverable as they are no longer preserved locally. SSD-Insider++ [9] further employed instant backup/recovery and lazy detection algorithms to mitigate the data loss

due to false negatives. However, the "lazy detection algorithm" still suffers from false negatives and, additionally, their design is only applicable to ransomware.

Guan et al. [20] proposed Bolt to enable system restoration after bare-metal malware analysis. However, Bolt is specifically designed to enable system restoration during the malware analysis, in which the malware analyst has a full control over the malware. It cannot be applied to our scenarios, in which the malware is out of the control of the victim, e.g., the malware may come anytime, and may behave arbitrarily. Chen et al. designed mobiDOM [14, 13] which aims to combat malware which comes any time. However, mobiDOM can only restore data to a historical state, rather than the exact state at the corruption point. In addition, mobiDOM relies on the malware detection which suffers from both false positives and false negatives.

8 Conclusion

In this work, we have designed MobiDR, the first secure data recovery system which can allow a victim mobile device to restore its data at the corruption point when suffering from malware attacks. Security analysis and experimental evaluation confirm that MobiDR can ensure recoverability of data at the corruption point, at the cost of a modest extra overhead.

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

References

1. Firefly AIO-3399J. https://en.t-firefly.com/product/industry/aio_3399.
2. Lpc-h3131. https://www.olimex.com/Products/ARM/NXP/LPC-H3131/.
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. TLS support in OPTEE #4075. https://github.com/OP-TEE/optee_os/issues/4075.
6. xxHash. https://cyan4973.github.io/xxHash/.
7. Mobile Malware. https://usa.kaspersky.com/resource-center/threats/mobile-malware, 1998.
8. SungHa Baek, Youngdon Jung, Aziz Mohaisen, Sungjin Lee, and DaeHun Nyang. Ssd-insider: Internal defense of solid-state drive against ransomware with perfect data recovery. In *Proceedings of ICDCS*, pages 875–884, 2018.
9. Sungha Baek, Youngdon Jung, Aziz Mohaisen, Sungjin Lee, and Daehun Nyang. Ssd-assisted ransomware detection and data recovery techniques. *IEEE Transactions on Computers*, 2020.
10. Marcel Breeuwsma, Martien De Jongh, Coert Klaver, Ronald Van Der Knijff, and Mark Roeloffs. Forensic data recovery from flash memory. *Small Scale Digital Device Forensics Journal*, 1(1):1–17, 2007.

11. Bo Chen and Reza Curtmola. Auditable version control systems. In *Proceedings of NDSS*, 2014.
12. Bo Chen, Reza Curtmola, and Jun Dai. Auditable version control systems in untrusted public clouds. In *Software Architecture for Big Data and the Cloud*, pages 353–366. Elsevier, 2017.
13. Niusen Chen and Bo Chen. Defending against os-level malware in mobile devices via real-time malware detection and storage restoration. *Journal of Cybersecurity and Privacy*, 2(2):311–328, 2022.
14. Niusen Chen, Wen Xie, and Bo Chen. Combating the os-level malware in mobile devices by leveraging isolation and steganography. In *International Conference on Applied Cryptography and Network Security*, pages 397–413. Springer, 2021.
15. Google Code. Opennfm. `https://code.google.com/p/opennfm/`.
16. 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 ACSAC*, pages 336–347. ACM, 2016.
17. C Chris Erway, Alptekin Küpçü, Charalampos Papamanthou, and Roberto Tamassia. Dynamic provable data possession. *ACM Transactions on Information and System Security (TISSEC)*, 17(4):1–29, 2015.
18. Ertem Esiner and Anwitaman Datta. Auditable versioned data storage outsourcing. *Future Generation Computer Systems*, 55:17–28, 2016.
19. Mohammad Etemad and Alptekin Küpçü. Transparent, distributed, and replicated dynamic provable data possession. In *International Conference on Applied Cryptography and Network Security*, pages 1–18. Springer, 2013.
20. 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 ACSAC*, pages 339–349, 2017.
21. Jian Huang, Jun Xu, Xinyu Xing, Peng Liu, and Moinuddin K Qureshi. Flashguard: Leveraging intrinsic flash properties to defend against encryption ransomware. In *Proceedings of ACM CCS*, pages 2231–2244. ACM, 2017.
22. Archanaa S Krishnan, Charles Suslowicz, Daniel Dinu, and Patrick Schaumont. Secure intermittent computing protocol: Protecting state across power loss. In *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 734–739. IEEE, 2019.
23. Donghyun Min, Donggyu Park, Jinwoo Ahn, Ryan Walker, Junghee Lee, Sungyong Park, and Youngjae Kim. Amoeba: an autonomous backup and recovery ssd for ransomware attack defense. *IEEE Computer Architecture Letters*, 17(2):245–248, 2018.
24. Pengfei Qiu, Dongsheng Wang, Yongqiang Lyu, and Gang Qu. Voltjockey: Breaching trustzone by software-controlled voltage manipulation over multi-core frequencies. In *Proceedings of ACM CCS*, pages 195–209, 2019.
25. 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*, pages 1–8. IEEE, 2017.
26. Sangat Vaidya, Santiago Torres-Arias, Reza Curtmola, and Justin Cappos. Commit signatures for centralized version control systems. In *IFIP International Conference on ICT Systems Security and Privacy Protection*, pages 359–373. Springer, 2019.

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. Peiying Wang, Shijie Jia, Bo Chen, Luning Xia, and Peng Liu. 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*, pages 327–338, 2019.
29. Xiaohao Wang, Yifan Yuan, You Zhou, Chance C Coats, and Jian Huang. Project almanac: A time-traveling solid-state drive. In *Proceedings of the Fourteenth EuroSys Conference 2019*, pages 1–16, 2019.
30. Ning Zhang, Kun Sun, Deborah Shands, Wenjing Lou, and Y Thomas Hou. Trusense: Information leakage from trustzone. In *Proceedings of IEEE INFOCOM*, pages 1097–1105. IEEE, 2018.
31. Yihua Zhang and Marina Blanton. Efficient dynamic provable possession of remote data via update trees. *ACM Transactions on Storage (TOS)*, 12(2):1–45, 2016.