Enabling Accurate Data Recovery for Mobile Devices against Malware Attacks

No Author Given

No Institute Given

Abstract. Mobile computing devices today suffer from various malware attacks. After the malware attack, it is challenging to restore the 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 [10]. 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 [18] 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 [18]. 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.

To enable data recovery, existing works either 1) purely rely on a remote version control server [23, 16, 22], or 2) purely rely on the local device [34, 35, 12, 13, 25, 29]. 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 either, because: First, FlashGuard [25] and TIMESSD [35] 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 for FlashGuard). This implies that the data which are not retained any more may become irrecoverable if compromised. Second, MimosaFTL [34], SSD-Insider [12], Amoeba [29] 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++ [13] tries to compensate the false negatives, but their strategy is specific for the ransomware and their lazy detection algorithm may still suffer from 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 will 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 [24, 25]: 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:

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

[1] Typically, delta data are committed to the remove server periodically rather than continuously, to reduce bandwidth/energy consumption.

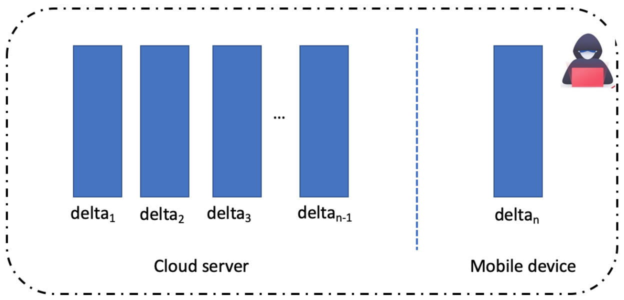


Fig. 1. A virtual version control system across the cloud server and the mobile device. We focus on defending against attackers in the mobile device side. Handling attackers in the cloud server side has been explored extensively before [23, 16, 21, 22, 37, 17, 32]

"freeze" the garbage collection over them. In other words, once the delta data are committed remotely, the garbage collection on them can runs normally.

Second, the 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, extracting delta data and committing them to the cloud server. Three issues need to be tackled. 1) How can the backup app securely extract delta data from the flash memory? The backup app runs in 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. 2) How can we prevent the malware from corrupting delta data sent from the FTL to the backup app? Our strategy is, the FTL will compute cryptographic tags for the delta data using a secret key, and each tag is computed over the data stored in a flash page; 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 generation process as well as the secret key. 3) 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, namely the tag verification and the committing process, into the TrustZone secure world, so that the backup app can correctly verify and commit the delta data based on the shared secret key.

Contributions. We summarize our contributions below:

– We have designed MobiDR, the first data recovery framework 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 is widely used in desktops, laptops, and mobile devices today. For instance, desktops and laptops increasingly use SSDs (solid state drive); main-stream smartphones and tablets use eMMC (MultiMediaCard) and UFS (Universal Flash Storage) cards; smart IoT devices use microSD cards. Flash memory typically includes NAND and NOR flash, and NAND flash is broadly used for mass storage. The entire flash memory is divided 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 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 turn "bad" and cannot be used to reliably store data.

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) a flash-based block device as external storage (e.g., an eMMC card, a microSD card, or a UFS card); and 2) Arm processors with TrustZone enabled. The flash memory is transparently managed by the *FTL*, exposing a block access interface. The Arm TrustZone can separate two worlds in the mobile device, a normal world running untrusted applications, and a *secure world* running *trusted applications.* 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 scaling 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 user data and asks for ransom, or trojan/backdoor malware which first steals user data and then corrupts them locally. We consider the strong OS-level malware [18] 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 com-

[2] Note that TrustZone has been broadly integrated into Arm Cortex-A processors or processors based on Armv7-A/Armv8-A architecture, as well as Arm Cortex-M processors which are built on Armv8-M architecture.

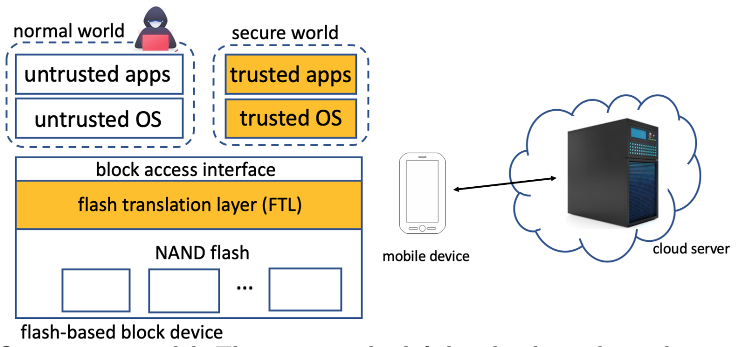


Fig. 2. Our system model. The part on the left-hand side is the architecture of the mobile device, in which the components in yellow color are isolated from the untrusted OS at the hardware level

mitted 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 data outsourced to an untrusted remote server has been explored extensively in prior works [11, 27, 21, 15, 16]. Other assumptions include: 1) TrustZone is secure, and the malware cannot compromise the secure world as well as trusted applications (TA) and data in it. This is a reasonable assumption in the domain of TrustZone technologies [24]. Although a few security leaks [36, 30] have been found in TrustZone, hardening TrustZone has been explored extensively in the literature [33] 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, changes of data (i.e. delta) in the local device will be committed securely to the cloud server (*the backup phase*). Upon a failure happens, MobiDR will retrieve the versioning data from the server, applying the most recent delta (stored locally) to restore the exact 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 as follows. 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 by the victim user to restore the data back to the *corruption point*. The malware is assumed to have been detected [18] at some point of time (i.e., the *detection point*) and removed from the victim device and, therefore, the recovery app can be run in the normal world. Before the detection point, the device has successfully conducted a backup process at some point recently (i.e., the *most recent backup point*). The recovery app will rely on the backup data in the remote server (correctness of the data is verified via the cryptographic tags), as well as the most recent delta preserved locally in the flash memory for data recovery. As the recovery app runs in the user space and cannot directly access flash memory, it needs to work with the FTL for recovery. An important step is to locate the corruption point, and we discuss two possible cases:

– If the malware starts to corrupt data after the *most recent backup point*, and is detected some time later (note that the detection algorithms usually rely on patterns generated by the malware, and such patterns would be extracted after the malware works for a while), the corruption point should be located somewhere between the most recent backup point and the detection point. For each new flash page written after the most recent backup point, the FTL will

[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, we should back up those data immediately and garbage collection can be immediately performed on this block.

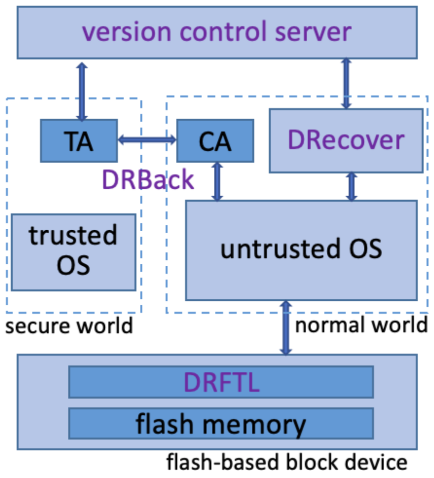


Fig. 3. The design overview of MobiDR

keep the corresponding timestamp; upon recovery, all the new flash pages will be sorted based on their corresponding timestamps. The recovery app can then conduct a binary search [34] between the most recent backup point (at this point, no new flash pages were written) and the detection point (at this point, the flash page with the largest timestamp was written) to efficiently locate the corruption point (detailed in Sec. 4.2).

– If the malware starts to corrupt the data before the *most recent backup point* but is detected later than it (e.g., the malware detection suffers from false negatives), the corruption point should be located somewhere between the initial point (i.e., when the device is first backed up) and the most recent backup point. The recovery app can locate the corruption point efficiently using a binary search between the initial point and the most recent backup point, as the data at any point are stored correctly in the remote server (detailed in Sec. 4.2).

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 after the malware has been detected [18] and eliminated. 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 [16, 23] 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 the backup phase are the most recent delta 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.

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[4] 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).

[4] 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.

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 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. After the malware is detected and eliminated, 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 similar to the backup phase, preventing them from being damaged in the recovery process (note that 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 work with the DRFTL to 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).

For case #1, a binary search method can be used to locate the corruption point. 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[5] 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. The binary search will be continued in either half, recursively. The number of user involvement will be $O(logl)$, 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.

For case #2, a binary search method can be also used to locate the corruption point. 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 data version contains corrupted data or not. If it has, the corruption point is located at a point even earlier; otherwise, the corruption point is located at a point later. The binary search will be continued in either half, recursively. After a historical version is located, the exact corruption point can be further located similar to case #1. The number of user involvement will be $O(logn + logl)$, 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.

After the recovery phase is finished, the DRecover will change the DRFTL back to the normal state.

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

5 Analysis and Discussion

5.1 Security Analysis

We *first* show that MobiDR can ensure: 1) any newly created delta can be correctly committed to the remote server upon each successful backup; and 2) the most recent delta data (not yet committed remotely) cannot be corrupted by the malware.

Any newly created delta can be correctly committed to the remote server upon each successful backup. 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 successfully to the remote server. Therefore, regardless of behaviors of the malware 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 user data will stay intact in the flash memory. Note that the DRFTL is transparent to the OS, and hence will not be affected by the OS-level malware and can always run correctly. The DRBack runs in the user space, and 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 of TrustZone and is responsible to verify the extracted data and send them to the remote sever once the verification is passed. The OS-level 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 which goes through the untrusted OS. This corruption attack can be mitigated since the DRFTL will compute cryptographic tags for the extracted data and, any corruption will be detected later by the TA, which is running in an isolated execution environment and 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 locally); others, the TA will send the extracted data, together with the associated tags, to the server. Meanwhile, the TA will send a confirmation to the DRFTL (via CA) that the backup process is done. Note that the confirmation is protected by the tag and, once it reaches the DRFTL, the DRFTL will know this data version has been backed up correctly. If the DRFTL receives a corrupted confirmation or has not received a correct confirmation for a long time, it should notify the user, since there is potentially a DoS attack here.

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.

We *then* show that MobiDR can always recover the data at the corruption point, and hence can achieve the data recovery guarantee. After the malware is detected and eliminated from the victim device, the system is in a healthy state. 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 the data at any point over the history, which surely includes the corruption point.

5.2 Discussion

How to share secret keys between the DRFTL and the TA. The DRFTL and the TA need to share secret keys. During the 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 a flash page and monitors the writes on this page; once the device owner writes the secret keys to this flash page, the FTL will read the page, copy the keys to other flash areas invisible to the OS, and securely clear this page.

About restoring the data after the corruption point. After the corruption point, the user applications may still work normally and create new data (due to the delay or the false negatives of the detection). Restoring user data after the corruption point would become a little tricky as it may be difficult to differentiate the data created by the user applications from that created by the malware. However, as MobiDR preserves all the historical data as well as the most recent changes, it is capable of restoring data even after the corruption point. A special case is that the malware writes itself as the new content to the device and potentially this “new content” may be committed to the cloud server. This would not be an issue. A more strict definition of the corruption point is the point of time when the system is clean, i.e., no malware. Therefore, when locating the corruption point upon recovery, the user should double verify that the restored system does not contain malware (note that user involvement is always necessary in DRecover).

Handling device failures. If the device fails (e.g., suffering from power loss [28]) upon backup and the most recent delta has not been committed yet, the user could try mobile device forensics [14] 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. In addition, this abnormal behavior will be informed to the user immediately.

6 Implementation and Evaluation

6.1 Implementation

We have implemented a prototype of MobiDR, which includes DRFTL, a DRBack app, a DRecover app, and a server program. The entire code base is available in [5].

DRFTL. To implement our DRFTL, we have modified OpenNFM [19], an open-sourced flash controller framework written in C. To keep track of the physical page locations of new delta data, the DRFTL reserves a small region in the flash memory and, once new data are written, it will keep the corresponding PBA in this region (organized as a stack) for the next backup process. The set of PBAs being kept is called "PBAs-to-extract". To prevent the delta data (invalidated by the malware) from being deleted before being backed up remotely, the DRFTL uses a modified garbage collection strategy so that the new delta data created since the last backup process will not be deleted. During the backup process, the DRBack app will continuously read a reserved LBA until all the new delta data have been extracted and, once the DRFTL receives a read request from this reserved LBA, it will pick a new physical page from PBAs-to-extract (i.e, the corresponding page PBA is popped from the stack), and return the delta data stored in it; the DRFTL also computes and returns a cryptographic tag (instantiated using HMAC-SHA1) over these delta data, the corresponding logical page address, as well as the version number and sequence number. During the recovery process, the DRFTL will work with the DRecover app to place the data back to their original LBAs.

DRBack. 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. The CA issues a (secret) special write sequence to the reserved LBAs (there is a fixed mapping between the sector addresses of the disk and the LBAs in the FTL, e.g., for 512-byte sector size and 2KB page size, each LBA address can be converted to the sector address by multiplying 4); the DRFTL monitors the FTL_Write interface and, once detecting this special write sequence, it will change its state to the backup state, and work with the DRBack to extract delta data from the raw flash memory. Once the delta data are completely extracted (the DRFTL will inform the CA by returning a special string), the CA will send them to the TA for verification and, if the TA successfully verifies them, it will send them to the version control server. Deployment of TA relies on the support of ARM TrustZone technology. OP-TEE (Open Portable Trusted Execution Environment) [6] is an open-source project which provides complete and thorough platform for developers to build their own applications on ARM TrustZone. There are many electronic boards which are officially supported by OP-TEE. We ported OP-TEE to Raspberry Pi 3B with Raspbian as the root file system. It should be noted that Raspberry Pi 3B did not support secure boot and other features [1]. However, as a prototype, Raspberry Pi is sufficient to demonstrate the feasibility of building the TA of DRBack. To enable the TA to communicate with the remote server, we relied on the TEE socket APIs.

DRecover. We have implemented DRecover in C. The DRecover app first extracts the most recent delta data from the flash memory (similar to the backup process), and then works with the remote version control server and the DRFTL to restore data at the corruption point. A significant step in DRecover is to commit an arbitrary historical version, retrieved from the remote server, back to the raw flash memory. This step is implemented as follows: 1) The DRecover app retrieves from the remote server a data version, and verifies it based on the associated tags. 2) The DRecover app issues a (secret) special write sequence to the reserved LBAs and, the DRFTL monitors the *FTL_Write* interface and detects this special write sequence. The DRFTL changes its state to the recovery state, and works with the DRecover to commit this entire data version back to the flash memory. Each data version is a collection of data chunks, each of which is corresponding to the data stored in a flash page and is associated with an LBA. Based on the LBA, the DRecover app can figure out the corresponding sector addresses (one page is corresponding to a few contiguous 512-byte sectors), and write this chunk to the corresponding sectors; the DRFTL will obtain this chunk from the *FTL_Write*, write it to a physical page, and update the mapping between the LBAs and the PBAs. 3) The DRecover app issues another special write sequence to the reserved LBAs, to terminate the recovery state.

We have also implemented a server program in C, which runs remotely and stores all the data sent by the mobile device in a version control manner. The server also allows the mobile device to retrieve an arbitrary data version.

6.2 Evaluation

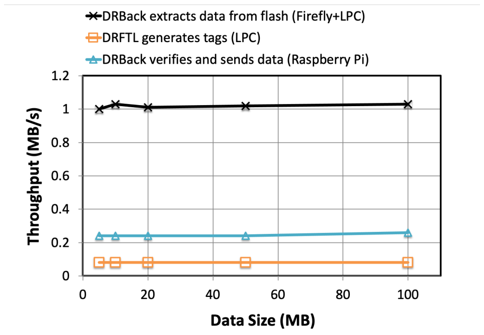
Experimental setup. To assess the performance of MobiDR in real-world mobile devices, we have ported our DRFTL to LPC-H3131 [4], a USB header development prototype board with ARM9 32-bit ARM926EJ-S (180Mhz), 32MB SDRAM, and 512MB NAND flash. After DRFTL is ported, LPC-H3131 can be used as a flash-based 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 [3] (Six-Core ARM 64-bit processor, up to 1.8GHz, Dual-Channel DDR3 - 1333MHz 4GB), which acts as the host computing device of the mobile device to perform I/Os on the flash storage provided by LPC-H3131. To measure the I/O throughput of MobiDR, we used a benchmark tool fio [2]. 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) [7], by porting OP-TEE[6] 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 set up in our lab (using NETGEAR RangeMax N150 WPN824N), and the electronic boards (Firefly AIO-3399J 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 [9].

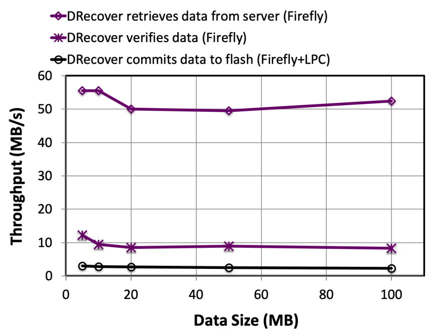
After having extracting the data, the DRBack will verify integrity of the data based on the associated tags, and send them (together with tags) to the remote server if the verification is successful. This sub-process is taken care by the TA of DRBack, which runs in the TrustZone secure world. As shown 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

[6] Currently OP-TEE has not supported TLS yet, which can be implemented as "a glue layer between mbedTLS and the GP API provided [8]".



(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

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

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

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, 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.

Assessing The Impact of MobiDR on The Flash Storage To support MobiDR, DRFTL needs to modify the original FTL in the flash-based block device. In the following, we assess the impact of our modification by measuring both the I/O throughput and the wear leveling effectiveness of the DRFTL, using the original FTL (OpenNFM) as a baseline for comparison.
I/O throughput. To assess the I/O throughput of DRFTL, we ran the benchmark tool fio in the host computing device, which performs I/Os on the external flash storage offered by LPC-H3131 (with DRFTL ported). We also assessed the

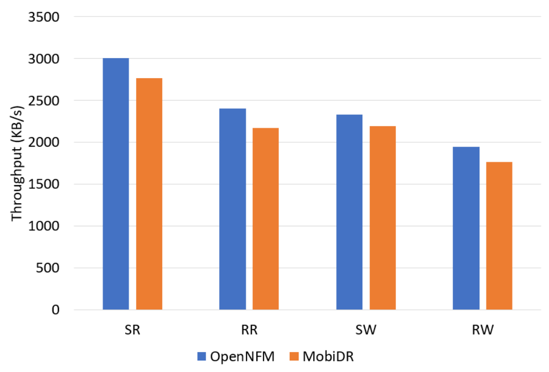


Fig. 5. Throughput comparison between MobiDR and OpenNFM. SR - Sequential Read; RR - Random read; SW - Sequential Write; RW - Random Write.

I/O throughput of the original OpenNFM using the same benchmark tool. The results are shown in Figure 5. We can observe that, compared to the original OpenNFM, DRFTL only slightly (i.e., less than 10%) decreases in I/O throughput under different access patterns. Compared to the original FTL, DRFTL requires extra operations on monitoring I/Os on the reserved LBAs for state changing (e.g., to change to a backup/ recovery state or to change back). However, those extra operations are lightweight and hence their impact on the I/O throughput is small.

Wear leveling. Wear leveling is important for prolonging the service life of flash memory. Therefore, we assessed the wear leveling effectiveness of DRFTL. The idea is to write a large amount of data and to evaluate how the erasures are distributed among the entire flash blocks. We calculated the wear leveling inequality (WLI) [26], and a small WLI value usually indicates a good wear leveling effectiveness. We filled the entire space of LPC-H3131 NAND flash, erased the entire device once completely filled (i.e., a test cycle). We repeated the aforementioned test cycle for 100 times under different wear leveling thresholds and computed each corresponding WLI. The results are shown in Table 3 and 4, respectively for MobiDR and OpenNFM. We can observe that: 1) The WLI values are pretty small for DRFTL, which indicate a good wear leveling effectiveness under different thresholds. 2) When the wear leveling threshold is decreased, the WLI value would decrease, indicating an increasing wear leveling effectiveness. This is because, a smaller wear leveling threshold would result in a more frequent wear leveling operation, hence a better wear leveling effectiveness (but the side effect is the average number of erasure counts per block will increase). 3) Compared to OpenNFM, MobiDR only decreases very slightly in WLI (approximately 3% decrease). This indicates that MobiDR only decreases the wear leveling effectiveness slightly, due to the extra operations needed to support secure data recovery.

Wear leveling threshold	Average #erasures (per test cycle)	WLI (%)
150	1.018	3.1
100	1.018	3.0
50	1.020	2.8

Table 3. WLI under different thresholds (MobiDR). A smaller threshold indicates a more frequent wear leveling operation

Wear leveling threshold	Average #erasures (per test cycle)	WLI (%)
150	1.018	3.0
100	1.018	2.9
50	1.019	2.7

Table 4. WLI under different thresholds (OpenNFM)

7 Related Work

Continella et al. designed ShieldFS [20], 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 [31], which hides the backup data to isolated storage space (via virtualization techniques) for data recovery. Both ShieldFS and RDS3 are not designed for combating the strong OS-level malware.

Huang et al. proposed FlashGuard [25] to enable data recovery from ransomware attacks even though the ransomware can compromise the OS. FlashGuard however, needs to preserve all the historical versions of "possibly" attacked data locally in the flash memory for a long period (e.g., 20 days [25]) to maximize probability of successful recovery, rendering the design impractical since the local storage space for computing devices is limited. SSD-insider (Baek et al. [12]), MimosaFTL (Wang et al. [34]), Amoeba (Min et al. [29]) further improved FlashGuard by incorporating ransomware detection into the FTL, so that the local device does not need to preserve those invalid data in the flash memory if it does not detect the ransomware. This may save local storage space, but malware detection unavoidably suffers from false negatives and, if a false negative happens, the data corrupted by the ransomware will become irrecoverable because they are no longer preserved locally in the flash memory. To improve SSD-Insider, SSD-Insider++ [13] 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.

Wang et al. proposed TIMESSD [35], which aims to enable data recovery by retaining past storage states in the local SSD. But still, due to the limited storage space in the local device, TIMESSD can only retain the history of storage states for up to eight weeks and cannot always guarantee recovery of the data at the corruption point. Guan et al. [24] proposed Bolt which enables system restoration after bare-metal malware analysis. In Bolt, the malware analyst has a full control of the malware, and knows where the corruption point is located. This significantly simplifies the recovery problem as the data can be simply restored to the state before malware analysis is conducted. However, Bolt can be only used for data recovery in the ideal malware analysis process.

8 Conclusion

In this work, we have designed MobiDR, a secure data recovery framework, 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.

References

1. OP-TEE documentation - Raspberry Pi 3. https://optee.readthedocs.io/en/latest/building/devices/rpi3.html.
2. fio. http://freecode.com/projects/fio.
3. Firefly AIO-3399J. https://en.t-firefly.com/product/industry/aio_3399.
4. Lpc-h3131. https://www.olimex.com/Products/ARM/NXP/LPC-H3131/.
5. MobiDR. https://github.com/emanresuretne/mobiDR.
6. Open Portable Trusted Execution Environment. https://www.op-tee.org/.
7. Raspberry Pi 3 Model B. https://www.raspberrypi.org/products/raspberry-pi-3-model-b/.
8. TLS support in OPTEE #4075. https://github.com/OP-TEE/optee_os/issues/4075.
9. xxHash. https://cyan4973.github.io/xxHash/.
10. Mobile Malware. https://usa.kaspersky.com/resource-center/threats/mobile-malware, 1998.
11. Giuseppe Ateniese, Randal Burns, Reza Curtmola, Joseph Herring, Lea Kissner, Zachary Peterson, and Dawn Song. Provable data possession at untrusted stores. In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 598–609, 2007.
12. 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.
13. Sungha Baek, Youngdon Jung, Aziz Mohaisen, Sungjin Lee, and Daehun Nyang. Ssd-assisted ransomware detection and data recovery techniques. *IEEE Transactions on Computers*, 2020.
14. 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.
15. Bo Chen and Reza Curtmola. Robust dynamic provable data possession. In *Distributed Computing Systems Workshops (ICDCSW), 2012 32nd International Conference on*, pages 515–525. IEEE, 2012.
16. Bo Chen and Reza Curtmola. Auditable version control systems. In *Proceedings of NDSS*, 2014.

17. 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.
18. 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.
19. Google Code. Opennfm. https://code.google.com/p/opennfm/.
20. 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.
21. 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.
22. Ertem Esiner and Anwitaman Datta. Auditable versioned data storage outsourcing. *Future Generation Computer Systems*, 55:17–28, 2016.
23. 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.
24. 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.
25. 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.
26. Shijie Jia, Luning Xia, Bo Chen, and Peng Liu. Deftl: Implementing plausibly deniable encryption in flash translation layer. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2217–2229. ACM, 2017.
27. Ari Juels and Burton S Kaliski Jr. Pors: Proofs of retrievability for large files. In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 584–597, 2007.
28. 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.
29. 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.
30. 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.
31. 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.
32. Sangat Vaidya, Santiago Torres-Arias, Reza Curtmola, and Justin Cappos. Commit signatures for centralized version control systems. In *IFIP International Con-*

ference on ICT Systems Security and Privacy Protection, pages 359–373. Springer, 2019.

33. 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.
34. 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.
35. 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.
36. 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.
37. 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.