APRON: Authenticated and Progressive System Image Renovation

Sangho Lee
Microsoft Research

Abstract

The integrity and availability of an operating system are important to securely use a computing device. Conventional schemes focus on how to prevent adversaries from corrupting the operating system or how to detect such corruption. However, how to recover the device from such corruption securely and efficiently is overlooked, resulting in lengthy system downtime with integrity violation and unavailability.

In this paper, we propose APRON, a novel scheme to renovate a corrupt or outdated operating system image securely and progressively. APRON concurrently and selectively repairs any invalid blocks on demand during and after the system boot, effectively minimizing the system downtime needed for a recovery. APRON verifies whether requested blocks are valid in the kernel using a signed Merkle hash tree computed over the valid, up-to-date system image. If they are invalid, it fetches corresponding blocks from a reliable source, verifies them, and replaces the requested blocks with the fetched ones. Once the system boots up, APRON runs a background thread to eventually renovate any other non-requested invalid blocks. Our evaluation shows that APRON has short downtime: it outperforms conventional recovery mechanisms by up to 28×. It runs real-world applications with an average runtime overhead of 9% during the renovation and with negligible overhead (0.01%) once the renovation is completed.

1 Introduction

Ensuring the integrity and availability of an operating system is crucial to the security of a computing device which is repeatedly threatened by adversaries. Specifically, the adversaries might compromise the operating system by exploiting unpatched vulnerabilities contained in its kernel, system applications, or shared libraries and, if exists, underlying systems software like a hypervisor. Then, they would permanently tamper with the system image (or files) stored in local storage to persist their control over the device (i.e., persistent malware [24, 29, 81]) or destroy it (i.e., destructive malware [53, 77, 82]). The computing device is no longer available in a valid form and demands *a recovery* as soon as the corruption is recognized [51, 98, 125].

Secure boot is a mechanism to boot a system while checking its integrity [8, 13, 52, 104, 124]. A trusted bootloader—whose authenticity is ensured by cryptography and hardware-based mechanisms [9, 102, 122]—measures (i.e., calculates a cryptographic hash over) the operating system and compares the measured value with an expected value before loading and passing control to the operating system. Any mismatch between them means that the operating system is in an invalid state. Specifically, the operating system might be (a) manipulated to embed a persistent backdoor, (b) destroyed to no longer work, or (c) downgraded to run a vulnerable old version. All these invalid states require an urgent fix.

Secure boot is suitable for securing devices with *image-based management*. They consistently deploy and update devices with *read-only* immutable system images built on the server-side and maintain their integrity on devices. However, such write protection is typically enforced within the kernel and adversaries can bypass it if they compromise the kernel [46, 58, 87]. Thus, operating systems require secure boot to verify the system image integrity, which is straightforward as expected measurement values are consistent or updated with coordination in image-based operating systems. Many modern operating systems for containers [36, 39, 45, 61, 86, 96, 105], Internet of Things (IoT) and edge devices [25, 37], mobile phones [6], mixed reality headsets [74], and personal computers [1, 10, 83, 117] adopt both.

System recovery is a logical next step when secure boot has found any corruption from a system image. Numerous security systems [3, 4, 17, 26, 51, 79, 98, 108, 125] even frequently reboot and recover (or reprovision) devices to protect them against persistent security and privacy threats and failures. However, to the best of our knowledge, all existing recovery mechanisms suffer from the following problems:

- **Downtime.** If a system recovery is necessary, a computing device enters a recovery environment [18, 75, 94, 120]. The recovery environment does not support any other regular tasks. That is, the system is *unavailable during recovery*, which is especially unacceptable to reboot-based security systems [3, 4, 17, 26, 51, 79, 98, 108, 125] and mission-critical tasks like edge computing [12, 84].

- **Inefficiency.** A recovery is inefficient because *it does not know which files or blocks it must fix in advance*. Existing mechanisms either (a) overwrite the entire system image to the local storage [40, 50, 51, 98, 125], which not only takes long but also is bad for storage lifetime [73, 103], or (b) verify each file or block to selectively fix corrupt one [54, 91], which is slower than the former.
- **Staleness.** A recovery typically relies on a system image backup stored in the local storage [50] which can be *outdated or corrupt*. To avoid this problem, it might fetch the latest system image from a reliable source, but this downloading prolongs the overall recovery time.

In this paper, we propose APRON, a novel approach to securely and progressively renovate an operating system image on a device. Unlike existing recovery mechanisms that fully repair a corrupt system image in a recovery environment and then boot into the recovered system, APRON securely boots into the system while repairing it, minimizing the downtime. It runs in the fresh kernel context to concurrently and selectively renovate any corrupt blocks which are requested during the system boot and after the startup of the operating system.

APRON *intervenes* between applications or kernel threads and the local storage containing a system image to verify and renovate invalid blocks on demand. To verify a requested block, APRON uses a Merkle hash tree [72] which is computed against an up-to-date system image and certified by an authorized entity (i.e., an administrator or an operating system vendor). Any hash verification error implies that a requested block is invalid (i.e., corrupt or outdated). Then, APRON renovates it by retrieving a corresponding block from a reliable source such as a remote server, verifying it, and overwriting it at the correct storage location. Once the system boots up, APRON additionally runs a background thread to renovate any other non-requested invalid blocks in the end. Also, it deduplicates any redundant network transfers.

We prototype APRON for Linux. We use device mapper [95] for intervening storage access and `dm-verity` [112] for hash tree verification. We also use Network Block Device (NBD) [22] as our remote storage protocol and its client, `nbdkit` [60], for userspace operations including HTTPS.

APRON ensures short system downtime and low runtime overhead. For a recovery with a 10 GiB system image in the servers with fast (local 1 Gbit/s) and slow (remote 100 Mbit/s) networks, APRON adds at most 5 s and 32 s to the downtime, respectively. It outperforms the full recovery by up to 28× and 12× and the delta recovery by up to 120× and 23×. APRON incurs an average runtime overhead of 9% on diverse real-world application tests from the Phoronix Test Suite [89] during renovation. Once the renovation is completed, the runtime overhead becomes negligible (0.01%).

In summary, this paper makes the following contributions:

- APRON is the first system that securely and progressively renovates a device's system image. It effectively minimizes the downtime due to recovery or update.
- APRON suggests a unified way to fix various types of invalid blocks including corrupt and outdated blocks.
- APRON shows its effectiveness (i.e., short downtime and low runtime overhead) in various configurations.

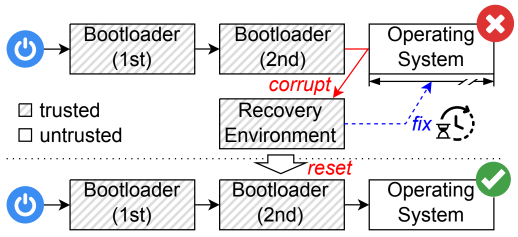


Figure 1: Secure boot with a normal recovery.

The source code of our prototype is publicly available at https://github.com/microsoft/APRON.

2 Background and Motivation

We will explain the background and motivation of APRON.

2.1 Secure Boot and Recovery

Secure boot ensures that a valid operating system will start to run on a device when it is powered on or reset. It has various synonyms, such as authenticated, measured, trusted, and verified boot, depending on security policies enforced or emphasized. Trusted or verified boot typically stops the boot procedure if it recognizes any verification failures, and initiates a recovery procedure. Authenticated or measured boot proceeds with the boot procedure while extending measurement values to a hardware component like Trusted Platform Module (TPM) [119] to report them to a system administrator for a later decision. In this paper, secure boot denotes trusted boot.

Figure 1 shows a procedure of secure boot with a recovery. When a device is powered on or reset, its CPU starts to execute the first-stage bootloader or boot firmware (e.g., Unified Extensible Firmware Interface (UEFI) [121], coreboot [30]) typically stored in a boot ROM. The first-stage bootloader verifies and loads the second-stage bootloader (e.g., GRUB [41], BOOTMGR [38]) stored in a specific location of local storage (e.g., EFI System Partition (ESP)). The second-stage bootloader verifies an operating system image stored in local storage. If the verification fails or a recovery has been requested, it boots into a recovery environment.

The recovery environment runs an agent program for recovery. The agent either downloads the latest system image from a known source and validates it or uses a local backup image to re-image the device [4, 18, 40, 50, 51, 98, 108, 125]. Finally, the agent reboots the device or directly loads the recovered kernel (using `kexec` in Linux [47] or Kernel Soft Reboot (KSR) in Windows [76]) and lets the kernel proceed the remaining boot procedure.

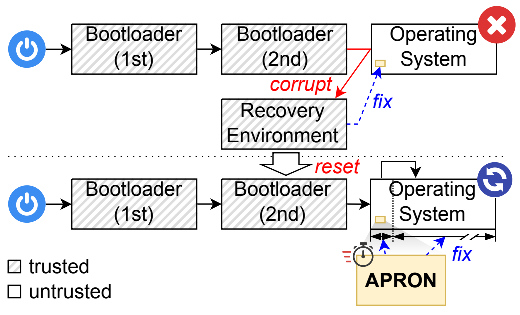


Figure 2: Secure boot with APRON.

2.2 Image-based System Management

Modern operating systems for specific use cases or casual end-users, such as container, IoT, edge, mobile phone, and personal computers [1, 6, 10, 25, 36, 37, 39, 45, 61, 74, 83, 86, 96, 105, 117] adopt image-based management to confine and simplify their deployment and update procedures. They split device storage into at least two different partitions including *read-only system partition* and *read-writable user partition*.

The system partition contains security-critical data which must not be modified, including kernel, drivers, and shared libraries. The kernel prevents any processes from modifying the partition [11, 112]. Administrators or operating system vendors generate or update a golden image for the system partition, sign it, and deploy it to devices. Instead of deploying the entire image, they can calculate and deploy the *delta* between the new and old images [34,70,91]. The device either fully overwrites the received image into its system partition or selectively updates it based on the delta. Later, a trusted bootloader verifies whether the system partition contains a valid, up-to-date system image before loading it.

The user partition contains casual applications and data populated by a user, which are not related to the device's critical operations. These user applications and data can be backed up by cloud storage, which is out of this paper's scope.

2.3 Motivation and Goal

Secure boot and image-based operating systems are widely deployed to real-world computing devices. However, their recovery mechanisms suffer from three important problems, motivating us to design a new approach that progressively recovers the device not only during its secure boot but also after the startup of its operating system (Figure 2).

Downtime for recovery. While the recovery environment repairs the system image, a computing device cannot conduct any regular operations that it is expected to do. That is, it cannot ensure a critical requirement, *availability*, for a long time. It is critical especially if the device leverages frequent reboots and recoveries to mitigate attacks and failures [3, 4, 17, 26, 51, 79, 98, 108, 125] or is deployed for mission-critical or time-sensitive tasks like edge computing [12, 84]. We cannot simply get rid of the recovery environment because we need a separate environment to securely trigger a recovery procedure at least. Instead, what we aim to achieve is

G1. Two-stage progressive recovery

Our approach progressively recovers the system image during and after the system boot. Its validity is ensured by a separate recovery environment.

Inefficient recovery due to unknown state. Existing recovery approaches are inefficient as they do not know which portions of the system image are corrupt in advance. Inspecting the entire image to identify invalid blocks and calculate delta takes long (§6.2). Instead, what we aim to achieve is

G2. State-aware on-demand recovery

Our approach identifies whether certain portions of the system image are corrupt and recover them on demand.

Insecure recovery due to stale image. Existing recovery approaches rely on a system image backup stored in local storage to recover the system image, which might be outdated or corrupted by adversaries. Fetching the latest image from a reliable source is a viable solution, but it is slow even if the image is compressed. Instead, what we aim to achieve is

G3. External up-to-date recovery

Our approach fetches authenticated portions of the system image on demand from a reliable external source.

3 Threat Model and Assumption

We consider how to recover a computing device from a remote adversary who can compromise the device's operating system including its kernel and system binaries as well as the underlying systems software (e.g., hypervisor and host operating system) if exists. The adversary can bypass the attack detection or prevention mechanisms for the device using unknown attack vectors, enabling initial compromise. After they compromise the system, they tamper with its storage to persist their control over it or corrupt it [24, 29, 53, 77, 81, 82].

We assume that the adversary cannot tamper with the boot firmware like other bare-metal recovery or reprovision systems [17, 51, 79, 108, 125]. Existing hardware components (e.g., boot ROM [66], TPM [119], and security coprocessors [9, 102, 122]) protect the boot firmware and its configuration data including the public key certificate of the authorized entity (i.e., administrator or operating system vendor) and a signed system version number for rollback prevention. We assume that a user or an administrator can recognize system compromise (e.g., by monitoring its external behaviors) and recover the system by forcefully rebooting it [14, 40, 118, 125] to let the boot firmware initiate a recovery procedure. In addition, the authorized entity generates and signs system images and metadata as well as operates servers for device management.

4 Design

In this section, we explain the design of APRON. It consists of (a) *storage layer* to authentically and progressively renovate the system partition during the system boot and after the startup of the operating system, (b) *server* to maintain and deploy valid, up-to-date system images, (c) *client* to fetch specific portions of the system image and deliver them to the storage layer for renovation, and (d) *recovery environment* to initiate APRON.

Storage layer. The APRON storage layer is responsible for on-demand renovation of invalid blocks (§4.2), background prefetching (§4.3), and deduplication (§4.4). All these tasks are securely performed based on a Merkle hash tree computed over a valid, up-to-date system image.

Server and client. The APRON server manages and updates operating system images, generates metadata for them (e.g., a Merkle hash tree), signs them, and deploys them to clients. The APRON client establishes a secure session with the deployment server to fetch data blocks based on the APRON storage layer's requests and deliver them to it (§4.5).

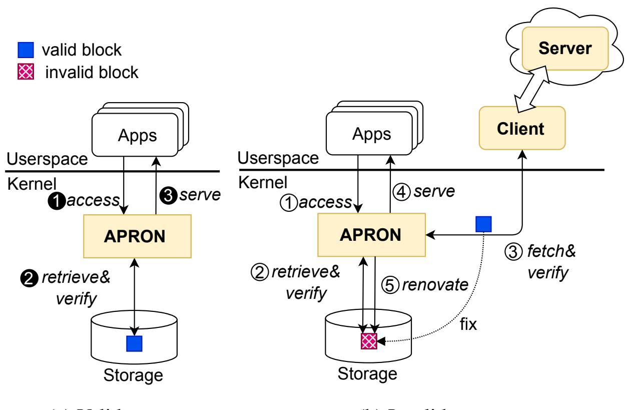
Recovery environment. The APRON-aware recovery environment prepares a minimal environment to initiate an operating system with APRON including operating system kernel and APRON metadata (§4.6).

4.1 Initialization

APRON is integrated into the operating system kernel and can be configured via boot parameters and APRON metadata. The APRON metadata includes a root hash value concatenated with a version number and Merkle hash tree calculated over the system partition. This metadata is prepared and signed by the APRON server (§4.5). APRON stores critical system files (e.g., kernel and device drivers) in the system partition and APRON metadata in a separate partition (to avoid circular dependency when calculating a hash tree). The APRON-aware recovery environment verifies this setting (§4.6).

During the system boot, APRON updates and verifies the APRON metadata. In particular, APRON attempts to download the latest APRON metadata from the APRON server. Then, it verifies the metadata's signature using the authorized entity's public key and checks whether the metadata's version number is greater than or equal to the reference version number. Both public key and reference version number are secured along with the boot firmware (§3). If the downloaded metadata is new and valid, APRON replaces the locally stored one with it and proceeds the system boot. If the version number in the new metadata is greater than the reference version number in the secure storage, APRON monotonically increases the reference version number accordingly as it means there is a legitimate update. Otherwise, APRON discards the downloaded one and proceeds the system boot with the local metadata.

When the operating system sets up a root filesystem, APRON prevents it from directly using the system partition. Instead, APRON places a *layer* (i.e., a virtual storage device or partition) over the system partition and makes the operating system use the storage layer as a read-only root filesystem. This layer allows APRON to intercept any accesses to the system partition, verify the individual accesses, and selectively fix the corresponding blocks using the Merkle hash tree.



(a) Valid access (b) Invalid access

Figure 3: On-demand renovation. Access to a valid block is verified and served using local storage (❶–❸). Access to an invalid block is served with the help of a remote server (①–⑤).

4.2 On-Demand Renovation

If any process or thread (except for APRON itself) requests a portion of the system partition, APRON transparently provides valid data either as they are or after renovating them. Figure 3 shows the overall on-demand renovation procedure. When a userspace application or kernel thread attempts to read data contained in the system partition via a filesystem or block device interface (❶), APRON retrieves a corresponding disk block expected to contain the requested data and verifies it using the Merkle hash tree (❷). If the retrieved block is valid (Figure 3a), APRON provides it to the requester (❸). To avoid repetitive storage retrieval and validation (i.e., to skip ❷), APRON maintains retrieved read-only blocks in an in-kernel cache and serves them to requesters later without re-validation until the blocks are evicted from the cache. Also, APRON identifies whether a requested block is a *zero block* (i.e., a data block filled with zeros) by checking the hash tree (i.e., a corresponding leaf node). In that case, it quickly provides a zero buffer to the requester without accessing the storage device at all. These two performance optimization methods are motivated by `dm-verity` [112].

If the retrieved block is invalid (Figure 3b), APRON fetches a corresponding block from the deployment server via the client and then verifies it (③). If the fetched block is valid, APRON provides it to the requester (④) to proceed with its execution immediately. Then, APRON overwrites the content of the fetched block into the corresponding location at the local storage device (⑤). If the fetched block is invalid due to network error, server error, or some other reason, APRON retries the block fetching up to a predefined number of times. If APRON fails to obtain the block eventually, it reboots the device to re-initiate the renovation.

In addition, APRON concurrently retrieves the same block from the local and remote storage (i.e., conduct ② and ③ in parallel) to effectively hide network overhead. Apron activates this concurrent fetching only if it confirms that the storage device is corrupt (i.e., it finds at least one invalid block during previous storage accesses) to avoid sending meaningless network requests.

4.3 Background Prefetcher

The on-demand renovation might fail if the network connection between a device and the deployment server becomes unreliable or slow during execution. To avoid such failures, Apron has *background prefetcher* that is a kernel thread to detect and renovate invalid blocks of the system partition in advance. Background prefetcher only inspects unidentified blocks which exclude verified or renovated blocks and zero blocks according to the hash tree. To support the former, Apron maintains a verified block bitmap and updates its bits when certain blocks are verified or renovated by either the on-demand renovation or background prefetcher.

Background prefetcher inspects unidentified blocks if it does not interfere with storage accesses from other applications or kernel threads. Specifically, it wakes up if there is no in-flight access to the local storage device, checks and renovates a limited number of unidentified blocks, and sleeps.

Background prefetcher detects consecutive invalid blocks and renovates them together (i.e., a batch renovation) to improve the renovation throughput. It is different from the on-demand renovation which repairs each urgently required block with low latency. When background prefetcher wakes up, it inspects the system partition from the first unidentified block according to the verified block bitmap to detect the first invalid block. Next, it inspects the following blocks until it encounters a valid block or the number of inspected blocks exceeds a threshold, resulting in a sequence of consecutive invalid blocks. Then, it fetches corresponding blocks from the remote storage together, verifies them, overwrites them to the local storage device for batch renovation, and updates the verified block bitmap accordingly. It uses an exponential backoff algorithm to dynamically adjust the threshold.

Disconnection. If background prefetcher has fully inspected the entire system partition, Apron can be completely disconnected from the deployment server until the device gets reset or it needs to renovate or update the system image (§4.6).

4.4 Deduplication

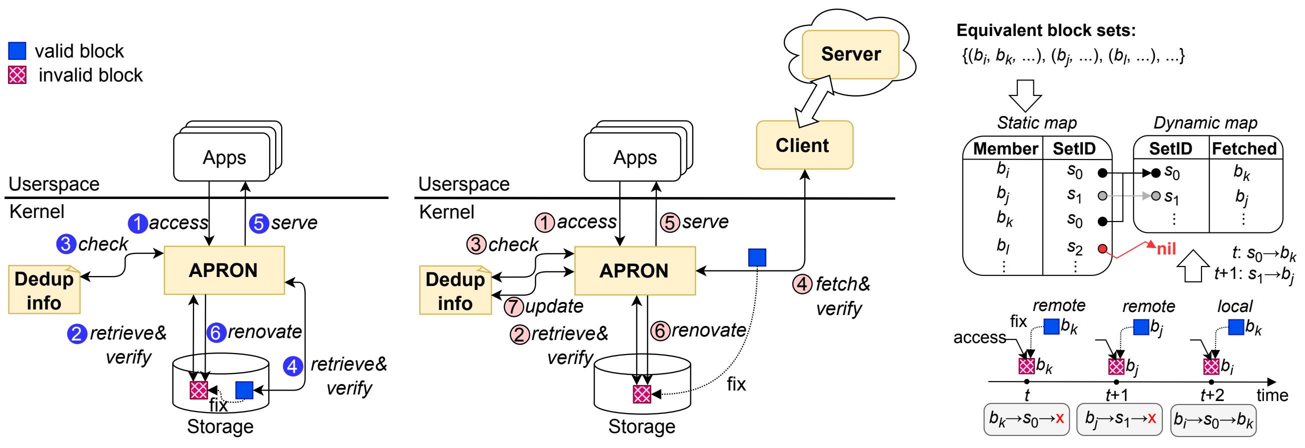
Apron might repetitively fetch equivalent blocks from the deployment server for renovation, which meaninglessly stresses both server and network. Apron avoids it using a data deduplication technique. Apron fetches a corresponding block from the server to renovate an invalid block only if it is unique or no other equivalent block of it has been fetched. If not, Apron uses the fetched equivalent block (in the local storage) for renovation. The Apron server creates deduplication metadata representing equivalent block sets in the system image. Specifically, the Apron server analyzes the system image (or the leaf nodes of its hash tree) to find equivalent blocks with the same content, forms sets by grouping equivalent blocks, assigns a unique identifier to each set, and adds this information to the deduplication metadata. The Apron server deploys and maintains the deduplication metadata together with the hash tree. To minimize the metadata size, Apron excludes unique blocks and zero blocks from it.

On the device, Apron maintains deduplication information consisting of two data structures for maintaining equivalent block sets and tracking whether and which block belonging to each set has been fetched, respectively. During initialization, Apron retrieves equivalent block set information from the deduplication metadata and constructs a *static block map* which associates each non-unique block (member) with its set identifier ($member \mapsto setID$). Later, Apron confirms whether a block is unique by looking up the static block map. If Apron fetches any non-unique block belonging to an equivalent block set for the first time (during the on-demand or background renovation), it adds this information to a *dynamic block map* to reversely associate the fetched block's set identifier with the fetched block ($setID \mapsto fetched$). This dynamic block map allows Apron to serve a duplicated block request using an equivalent block stored in the local storage.

Figure 4 shows the on-demand renovation with deduplication. The deduplication neither affects access to any valid blocks nor any invalid but unique blocks. Thus, we do not re-explain them (refer to §4.2). Instead, we focus on access to an invalid non-unique block that might have an equivalent block fetched and stored in the local storage device. If Apron confirms that a requested invalid block is not unique and its equivalent block is stored in the local storage device according to the deduplication information (i.e., the dynamic block map) (Figure 4a ❸), Apron retrieves the equivalent block from the local storage device and verifies it (❹). If the retrieved equivalent block is valid, Apron provides it to the requester (❺) and fixes the invalid local block (❻). In addition, it skips ❹ if the equivalent block is in the cache.

If the requested invalid block has no fetched equivalent block or the retrieved equivalent block is invalid (Figure 4b), Apron fetches a corresponding block from the deployment server and verifies it (④), provides it to the requester (⑤), and renovates the invalid local block (⑥). Apron updates the deduplication information (⑦) by adding the requested block as a fetched one to the dynamic block map.

Figure 4c depicts how deduplication works. In the beginning, Apron constructs a static block map using equivalent block sets and an empty dynamic block map. At time t, a thread requests an invalid and non-unique block b_k with the set ID of s_0. Apron fetches a corresponding block from the deployment server since the dynamic block map has no entry for s_0 and adds $s_0 \mapsto b_k$ to the dynamic map. At time $t+1$, a thread requests b_j with the set ID of s_1. Again, Apron fetches a corresponding block from the server and adds $s_1 \mapsto b_j$ to



(a) Invalid access with fetched equivalent block (b) Invalid access without fetched equivalent block (c) Deduplication management

Figure 4: On-demand renovation with deduplication. Access to an invalid block with a local equivalent block is served and renovated using the equivalent block (❶–❻). Access to an invalid block without a local equivalent block is served with the help of a remote server while renovating the requested block and updating the deduplication information (①–⑦).

the dynamic map. At time $t+2$, a thread requests b_i with the set ID of s_0. This time, APRON retrieves b_k from the local storage to renovate b_i because the dynamic map has $s_0 \mapsto b_k$.

Background prefetcher with deduplication. Background prefetcher prioritizes unique or never-fetched invalid blocks which must be remotely renovated over other invalid blocks which can be locally fixed. To this end, it treats invalid non-unique blocks as semi-valid if their equivalent blocks have been fetched and does not renovate them urgently. At the end of storage inspection, it spawns another kernel thread, *duplicator*, to renovate all semi-valid blocks using their local equivalent blocks. Like background prefetcher, duplicator wakes up if there is no in-flight access to the local storage device, renovates a single set of consecutive blocks, and sleeps.

4.5 Server and Client

The APRON servers are responsible for two major tasks: generating and maintaining operating system images as well as their associated metadata; and deploying them to the APRON client running in computing devices via a secure channel.

Management server. The management server is a trusted server operated by administrators or operating system vendors. It generates APRON metadata when a system image is newly generated or updated. First, it analyzes the system image to zero out unallocated data blocks—to handle sparse images—and then calculates a hash tree over the image. Next, it generates deduplication metadata (i.e., figures out equivalent block sets) using the hash tree's leaf nodes. Then, it monotonically increases the hash tree version number, and signs a concatenation of the root hash value and version number. Lastly, it stores the system image and APRON metadata in a dedicated place to be accessible to the deployment server.

Deployment server. The deployment server provides system images and APRON metadata to APRON devices. It interacts with devices via secure channels (i.e., TLS) to prevent external entities from manipulating or eavesdropping on communication. To handle data block requests, it uses a remote storage protocol (e.g., iSCSI, NBD) or a regular data transfer protocol (e.g., HTTP, FTP). The former lets it efficiently and transparently handle requests but is bad for portability. The latter introduces overhead including bloated packet headers and extra translation but ensures portability and proximity (e.g., with Content Delivery Network (CDN)). They respectively have pros and cons, and which one we are expected to choose one of them according to system configurations.

Client. The APRON client is a userspace service that interacts with the deployment server. It establishes a secure channel with the server and uses either a remote storage protocol or a regular data transfer protocol to receive data blocks.

Unlike the management server, APRON does not trust both the deployment server, which could be operated by a third party like a CDN, and the userspace client, which might be compromised by an attacker. The in-kernel APRON storage layer always verifies received blocks with the versioned APRON metadata signed by the management server. Even if the deployment server or the userspace client arbitrarily tampers with or downgrades the system image or metadata, the in-kernel APRON layer identifies such attempts based on signature and version mismatch.

4.6 First-Stage Recovery and Update

APRON can start to work only if a device has critical system files (i.e., kernel and device drivers for storage and network) and APRON metadata in valid forms. APRON relies on a first-stage recovery environment to ensure it. If any of them are invalid, the trusted bootloader enters the recovery environment to obtain their latest versions from the deployment server. A conventional APRON-unaware recovery environment would fully download both critical system files and APRON metadata from the deployment server via HTTPS. In contrast, an

APRON-aware recovery environment only needs to download the metadata via HTTPS while selectively renovating invalid portions of the critical system files via APRON. It is possible because a recovery environment typically shares the same (or minimized) kernel with the main operating system, so we modify its kernel to incorporate APRON.

Scheduled update. APRON works as an update mechanism for non-compromised operating systems. If APRON recognizes an update during system execution, it stages updated APRON metadata on the APRON partition. Through a reset, the recovery environment replaces APRON metadata with the staged one. Finally, APRON progressively updates the system during its execution. Unlike existing update mechanisms [5,7,16], APRON does not need to reserve extra storage to temporarily store a (potentially large) update file and apply multiple update files in a proper order.

5 Implementation

In this section, we explain how we develop APRON for Linux.

Initialization. We use `initramfs` to configure and initialize the APRON storage layer as the root filesystem. Our `initramfs` checks whether the signature and version number of a given root hash value are valid using the public portion of our signing key and reference version number we provision to TPM NVRAM indexes. If they are valid, it initializes the APRON storage layer and mounts it at a specific point. Further, it creates a `tmpfs` filesystem to use it as a writable overlay for the storage layer using `overlayfs` [23] to support applications that only work with a writable root filesystem. Lastly, it sets up the overlaid storage layer as the root filesystem. We store the `initramfs` in the system partition, so it is secured by APRON as well.

Storage layer. We implement the APRON storage layer as a loadable kernel module written in approximately 1,200 lines of C code on Linux kernel version 5.11. The storage layer prototype consists of two virtual block devices representing local and remote block devices, respectively.

The local virtual block device intervenes with any access to the system partition and is exposed as a regular block device to the outside (to work as the root filesystem). It is based on the device mapper framework [95]. The storage layer verifies all accesses to the local block device using a Merkle hash tree based on `dm-verity` [112]. It also spawns background prefetcher as a kernel thread to inspect the local block device while maintaining and using the deduplication information. If the storage layer finds any corrupt blocks from the local block device, it renovates them by copying corresponding blocks from the remote virtual block device to the local virtual block device while verifying them using the hash tree. That is, it securely makes the content of the local block device equivalent to that of the remote block device. This approach also allows APRON to use a local backup device for renovation instead of a remote device if the network condition is bad or a new system image is buggy. Background prefetcher uses `kcopyd` [114] to efficiently copy a sequence of data blocks between block devices. In addition, APRON's every storage access is cached by the `dm-bufio` interface.

The remote virtual block device interacts with the deployment server via the APRON client based on NBD [22]. We use NBD because it is easy to configure (both in client and server) and has a small code base. If needed, APRON can work with other advanced remote block storage such as Ceph [123] for better efficiency, reliability, and scalability.

Server. The APRON server has a program to identify equivalent block sets written in 170 lines of Rust code, Bash scripts to automate the creation, management, and deployment of system images and APRON metadata, and other server applications. It uses `zerofree` [126] to zero out the unallocated blocks of system images, `veritysetup` [112] to calculate Merkle hash trees over them, and `openssl` [115] to sign the root hash value concatenated with a version number. Also, it uses `nbdkit` [60] and `lighttpd` [65] to operate an NBD server with TLS and an HTTPS server, respectively.

Client. We prototype the APRON client using `nbd-client` [32] and `nbdkit` [60]. It uses `nbd-client` to connect to the APRON server via NBD over TLS and configure this session as the storage layer's remote block device. Also, to interact with the APRON server via HTTPS, it uses `nbdkit` to spawn a device-local NBD server backed by the HTTPS server and lets `nbd-client` connect to this local NBD server (via a Unix domain socket). `nbdkit` relies on its `curl` plugin to fetch specific portions of system image files from the HTTPS server using HTTP `range` requests.

Recovery environment. The APRON recovery environment is based on the Linux kernel with APRON and `initramfs`. It includes `curl` [107] to download APRON metadata from the APRON server, and both `nbd-client` and `nbdkit` to renovate essential system files (i.e., kernel and `initramfs`) in the system partition on-demand. Unlike the main operating system, we decide not to spawn background prefetcher in the recovery environment because it only runs for a short amount of time. In addition, it contains `kexec` [47] to directly load the renovated operating system's kernel.

Bootloading. We decide not to manipulate the first-stage bootloader (UEFI [121]) of our computing device (e.g., replace it with coreboot [30] or replace its platform key with our own key [52]) because it is too intrusive and does not affect the core functionalities of APRON. Instead, we rely on the current UEFI-based Linux boot procedure that securely loads the second-stage bootloader, GRUB [41], signed by Linux vendors (i.e., Canonical in our case) through Shim [68] signed by Microsoft. We modify GRUB's configuration to make it load either the operating system with APRON or the APRON recovery environment.

6 Evaluation

We evaluate APRON by answering the following questions:

- **RQ1.** Does APRON ensure short system downtime when it needs to renovate the system during boot? (§6.2)
- **RQ2.** How much overhead does APRON add to other workloads during renovation? (§6.3)
- **RQ3.** What is the network usage of APRON for renovation? (§6.4)
- **RQ4**. Does APRON complete renovation within a reasonable time? (§6.5)

6.1 Setup

Device. We use a desktop computer featuring an Intel Core i5-8500 CPU (six cores) at 3 GHz, 8 GiB of RAM, and 1 TB of PCIe 3.0 NVMe SSD as an APRON device.

Server. The APRON device frequently downloads small data packets to selectively renovate the system image, subject to network performance. To evaluate it, we use two APRON servers with fast and slow network configurations. The fast-network server is a mini computer connected to the 1 GbE switch that the APRON device is also connected to. It features an Intel Pentium Silver J5005 CPU (four cores) at 1.5 GHz, 8 GiB of RAM, and 500 GB of SATA SSD. The slow-network server is a Virtual Machine (VM) in Microsoft Azure. It features two Intel vCPUs at 2.6 GHz, 8 GiB of RAM, and 30 GiB of Premium SSD. According to `netperf` [49], the median TCP latencies between the device and two servers are 0.24 ms and 5.45 ms, and the TCP throughputs between them are 934 Mbit/s and 931 Mbit/s, respectively. We additionally throttle the slow-network server's bandwidth to 100 Mbit/s to evaluate low-throughput cases.

Configuration. We install Ubuntu Server 20.04 on the device while replacing its kernel and modules with ours, computing a hash tree over the system partition, and changing its GRUB configuration. We do not use existing image-based operating systems because they are highly customized for specific platforms (e.g., VM, mobile phone). We reserve 10 GiB for a device's system partition formatted with `ext4`. Ubuntu Server 20.04 occupies 5.5 GiB of the system partition. It becomes 1.6 GiB with `gzip`. We also reserve 100 MiB to store the APRON metadata. We use 4 KiB as the data and hash block size and SHA-256 for constructing the hash tree, and RSA-4096 to sign the root hash. The sizes of the hash tree and deduplication metadata are 81 MiB and 1.6 MiB, respectively. We install Ubuntu Server 20.04 to our servers. We repeat all experiments at least 10 times and report their average values except for benchmark tools with internal repetitions (§6.3) and an experiment with deterministic results (§6.4).

6.2 System Downtime (RQ1)

To minimize system downtime, APRON boots into a system while renovating its invalid blocks requested during the boot. We compare it against existing recovery mechanisms which repair all invalid blocks before the system boot. In general, both attacks and legitimate updates change a portion of the system image (which will be explained later), but what and how many blocks they will change are unpredictable. Thus, we randomly corrupt 1%–100% of the system partition for evaluation (i.e., zero some of or all its 4 KiB blocks). Whether we use zero or non-zero blocks does not affect APRON's performance (§6.6). We measure the delay solely introduced by APRON; that is, we exclude any other delays due to hardware initialization, bootloader loading and execution, and operating system loading, which are ∼16 s in total on our device, because they are independent of APRON. We use `systemd-analyze` [64] for this measurement. In addition, we omit the case without corruption because APRON does not delay it.

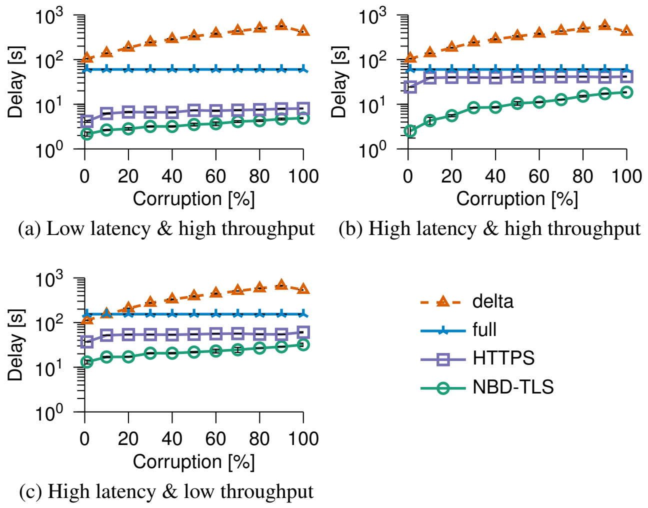


Figure 5: Boot-time delay comparison between APRON and other recovery mechanisms.

Full recovery. For comparison, we implement a full recovery mechanism like existing mechanisms [4, 40, 50, 51, 98, 108, 125]. In a recovery environment (i.e., before the system boot), an agent downloads the compressed image via HTTPS while concurrently decompressing it to the local storage. Then, it boots into the recovered system. We omit additional image validation because we trust TLS. The recovery takes ∼60 s (high throughput) and ∼154 s (low throughput), which is 4× and 10× longer than the system boot time. It is independent of the corruption ratio and marginally affected by network latency.

Delta recovery. We implement a delta recovery mechanism using `rdiff` [91] that the SWUpdate project [16] uses. `rdiff` consists of (a) signature computation, (b) delta computation, and (c) patch adoption. Delta update is efficient because it can pre-compute (a) and (b) on the server-side, but the delta recovery cannot leverage such pre-computation (details are in Appendix B.) In total, it takes 105–561 s (high throughput) and 112–665 s (low throughput), which is 7–35× and 7–42× longer than the system boot time. It depends on the corruption ratio and is marginally affected by network latency. The delta recovery is slow, but it reduces network traffic (31 MiB–1.6 GiB) and unnecessary storage writes.

APRON. Figure 5 shows APRON's boot-time delay while varying the network latency, network throughput, and corruption ratio. The delay is 2.2–4.9 s (NBD over TLS) when latency is low and throughput is high, which is 14%–31% of the boot time. It becomes 0.2–1.2× longer than the boot time when latency increases and 0.8–2.0× longer than the boot time when latency increases and throughput decreases. They are up to 27.8×, 23.9×, and 11.8× shorter than the full recovery, and up to 119.8×, 41.6×, and 23.1× shorter than the delta recovery, respectively. In addition, as expected, APRON with HTTPS is 1.6–9.7× slower than APRON with NBD over TLS due to extra translations.

Summary. APRON ensures short downtime because it apparently repairs the blocks required to boot the system first and the remaining blocks later once the system boots up. It outperforms existing mechanisms especially when (a) the network latency is low, (b) the network throughput is high, and (c) the number of invalid blocks is small. These conditions are satisfied in practice. The network performance is improving. Also, the number of invalid blocks is generally small in both attack and update cases. For example, a persistent backdoor usually consists of a few small executables [29, 81]. We further analyze how Flatcar Container Linux [39] (an image-based operating system) and Fedora Cloud [93] (a pre-installed cloud image) change over their releases. Across 36 Flatcar releases for two years (version 2512.2.0 to 3139.1.2 for QEMU x64), 9.7%–12.2% of their 8.5 GiB images change. Also, across 11 Fedora Cloud releases for five years (version 26 to 36 for QEMU x64), 12.5%–21.7% of their 4.5 GiB images change. Thus, APRON is effective in decreasing the system downtime.

6.3 Runtime Overhead (RQ2)

APRON verifies and renovates the (remaining) system partition during system execution. We evaluate how its **verification** and **renovation** processes affect the runtime performance of other workloads using benchmark programs. We ensure the benchmark programs terminate before renovation is completed. Otherwise, they run on the recovered system which hides renovation overhead. To this end, we prolong renovation using a fully corrupt system image and the slow-network server and select benchmark programs which take shorter than the renovation (§6.5). We install them in another system image copy and execute them via APRON. We compare APRON to a **pristine** environment with the same hardware and operating system except that it runs an unmodified kernel.

Microbenchmark. Figure 6 shows LMbench [71] system call execution times normalized to those from the pristine environment. Both in a verification condition and during the renovation, the overheads of the APRON device over the pristine device are 0%–40%. As expected, APRON affects system calls related to filesystem and network (e.g., `stat`, `fstat`, `open`/`close`, UNIX socket). On a geometric mean, the overheads of the APRON device in a verification condition and during the renovation are 7% and 8%, respectively.

The APRON device might download a lot of data from the deployment server to renovate the system partition, so it might affect the network performance of other applications. We use LMbench's network throughput evaluation results to identify whether and how APRON affects LMbench's network throughputs (Figure 7). In a verification condition, the network throughput of LMbench on the APRON device is 3.6% (geometric mean) lower than that on the pristine device. During the renovation, the network throughput of LMbench on the APRON device is 11.8% (geometric mean) lower than that on the pristine device. Consequently, APRON's renovation noticeably affects the network throughput of other applications only during renovation. In addition, if we turn off APRON's verification once the renovation is completed, the overhead becomes almost zero.

Macrobenchmark. We evaluate APRON with 11 real-world application tests from Phoronix Test Suite [89] (Figure 8). The overheads of APRON during renovation over the pristine environment are 1.9%–21% (geometric mean: 9%). As expected, renovation affects I/O-intensive workloads (e.g., 7-Zip, Apache, and Memcached) whereas less affects compute-intensive workloads (e.g., Crafty, FLAC, and PyBench). Without renovation (i.e., the verification condition), APRON's overhead over the pristine environment is negligible (0.01%).

6.4 Network Usage (RQ3)

During renovation, APRON fetches blocks required for recovery from the deployment server. We evaluate this network usage. We use the randomly corrupt system images again (§6.2). We count the number of bytes the server sends to the APRON device at the server (using `lighttpd` logs) while enabling APRON's optimization (i.e., zero block ignorance and deduplication). This network usage is independent of network latency and throughput.

Figure 9 shows how many bytes APRON downloads from the server while varying the corruption ratio. It downloads 44.6%–94.0% of the corrupt blocks, which are only 1.1–2.9× larger than corresponding `rdiff` deltas. APRON's optimization is effective especially when the number of invalid blocks is large. This is because it increases the probability that multiple invalid blocks are equivalent such that no repetitive downloads are needed due to deduplication (§4.4).

6.5 Complete Renovation Time (RQ4)

Once the system boots up, APRON performs both on-demand and background renovation. We measure how long it takes to complete the renovation when the system is idle or busy. Hastening the complete renovation is not our goal and that is why we assign a low priority to background prefetcher to minimally affect other workloads (§6.3). We compare it against the full and delta recovery to check whether it is reasonable.

Idle system. Figure 10 shows APRON's complete renovation time in an idle system while varying the network latency, network throughput, and corruption ratio. The complete renovation (NBD over TLS) demands 47.4–104.5 s when latency

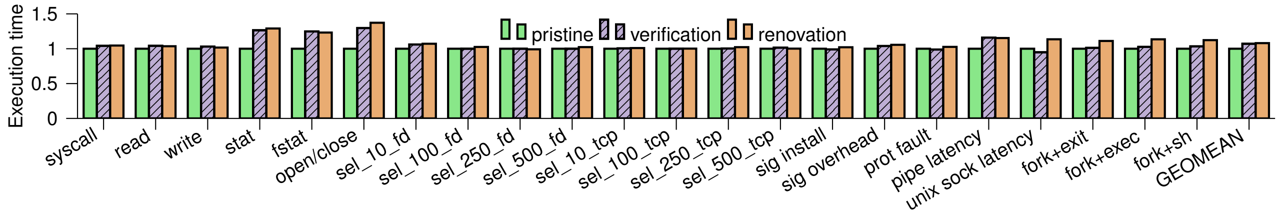


Figure 6: LMbench system call execution time normalized to the pristine cases (**pristine**: Ubuntu Server 20.04 without APRON; **verification**: APRON without a renovation process but with hash-tree verification; **renovation**: APRON with a renovation process).

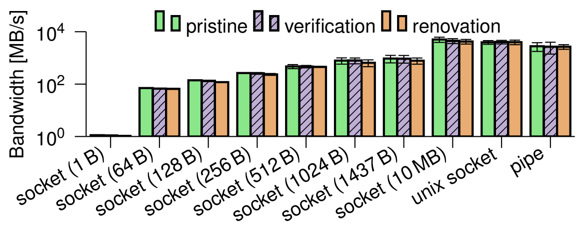


Figure 7: LMbench network throughput.

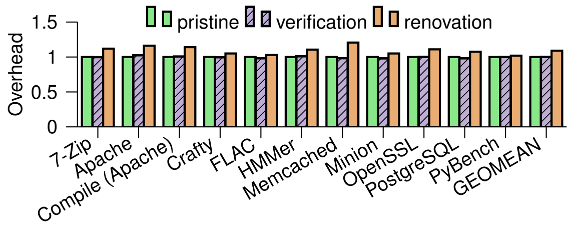


Figure 8: Normalized Phoronix Test Suite overhead.

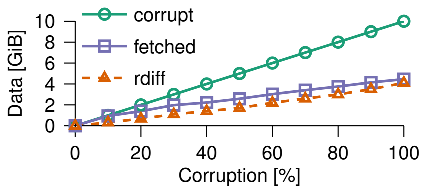
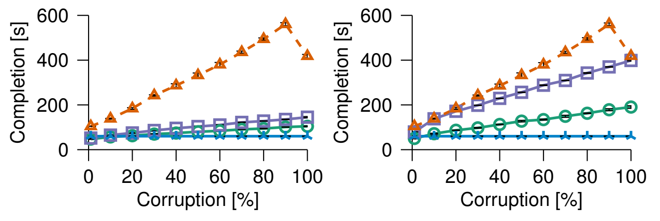
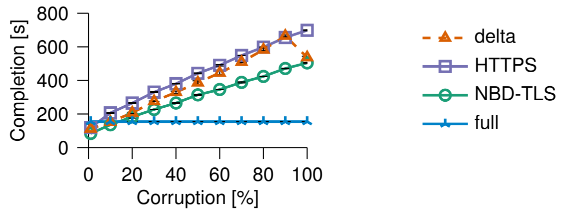


Figure 9: Network usage of APRON.



(a) Low latency & high throughput (b) High latency & high throughput



(c) High latency & low throughput

Figure 10: Complete recovery time comparison between APRON and other recovery mechanisms.

is low and throughput is high, 50.2–190.4 s when latency increases, and 84.0–504.3 s when latency increases and throughput decreases. They take 0.8–1.7×, 0.8–3.2×, and 0.5–3.3× longer than the full recovery, and 2.2–5.5×, 1.9–3.1×, and 1.1–1.4× shorter than the delta recovery, respectively. In addition, APRON with HTTPS takes 1.1–2.1× longer than APRON with NBD over TLS. APRON's complete renovation is subject to network latency and throughput because it does not benefit from bulk network transfer and compression.

Busy system. We make the system busy by running the Memcached test from Phoronix [89], which heavily contends with APRON (Figure 8), once the system boots up. The complete renovation is delayed by at most 7.0% (low latency) and 2.5% (high latency and low throughput). Overall, the renovation is moderately affected by the system's busyness. The renovation with the slow-network server suffers less from the busyness than that with the fast-network server since the slow network performance dominates the renovation overhead.

Summary. Although APRON renovates the system during its execution in the background, it finishes the renovation within a reasonable time—i.e., it is at most 3× slower than the full recovery. Further, it is comparable to or even faster than the full recovery if the network is fast and the number of invalid blocks is small. Both are satisfied in practice (§6.2).

6.6 Miscellaneous

Non-zero corruption. We confirm whether we use zero or non-zero blocks to corrupt the system image does not affect APRON's performance. If a block is expected to be a zero block, APRON does not touch it regardless of corruption. If not, APRON renovates it regardless of whether it is overwritten by a zero or non-zero erroneous block. Non-zero corruption slows down the delta recovery as it complicates `rdiff` signatures, but showing its worst case is not our interest.

Deduplication. The deduplication decreases not only network usage but also renovation time to some extent. Without it, the renovation is delayed by up to 2.2% (low latency) and 9.0% (high latency and low throughput). The renovation with the fast-network server marginally benefits from the deduplication because it aims for reducing the network overhead (§6.4). In contrast, the system downtime does not benefit from the deduplication because only a few blocks are requested during the system boot.

Memory usage. APRON uses at most ∼120 MiB of additional memory when it renovates a fully corrupt system image to cache fetched blocks and maintain its data structures. This memory is reclaimed once the renovation is completed.

7 Security Analysis

In this section, we analyze the security of APRON. We focus on how APRON mitigates persistent kernel attacks. Also, we explain how APRON detects or prevents less severe but frequent userspace and network attacks.

Kernel attack. APRON efficiently recovers a computing device from advanced adversaries who have compromised the device with critical kernel vulnerabilities. APRON runs in the kernel and it does not assume any other non-standard kernel integrity protection or monitoring technologies [15, 44, 69, 78, 88, 128] to protect itself. To this end, if adversaries compromise the kernel, they can *temporarily* deactivate APRON and corrupt the system image. However, APRON eventually defeats them based on a system administrator's input: the administrator can detect a misbehaving device using device or network monitoring tools and forcefully reset it using existing, standardized techniques [40, 118, 125]. This reset eliminates in-memory exploits and activates APRON via secure boot again. If the administrator prepares an updated image fixing the exploited vulnerability, APRON prevents the adversaries from reusing the vulnerability by rapidly recovering the system with the updated image. Even if no patch is prepared, APRON causes hardship to the adversaries since they must repeatedly compromise the operating system across forceful device resets to persist their control or system destruction. Such repetitive attack attempts are highly visible and thus can be detected and mitigated by network-level techniques [3].

Userspace attack. APRON prevents or detects userspace attacks against it through all four attack surfaces userspace attackers can access (§5): (a) filesystem containing operating system files, (b) APRON storage layer, (c) storage device storing the system image, and (d) APRON client. First, APRON prevents or detects the modification of its filesystem. APRON mounts the filesystem for the operating system as read only, preventing non-privileged attacks. Adversaries with a root privilege can remount the filesystem and modify it. However, such modifications only remain in memory and are not reflected in the underlying write-protected APRON storage layer. Second, APRON prevents the modification of its storage layer. The APRON storage layer ignores any block write requests via block IO interfaces. Thus, both non-privileged and privileged adversaries cannot modify it. Third, APRON prevents or detects the modification of its storage device. Non-privileged adversaries cannot access the storage device. In contrast, privileged adversaries can tamper with the storage device using block IO interfaces. However, the APRON storage layer identifies and reverts such manipulation based on the signed hash tree. Fourth, APRON detects a misbehaving APRON client. Privileged adversaries can compromise the userspace APRON client to deliver manipulated data to the APRON storage layer. However, APRON identifies and ignores such manipulated data based on the signed hash tree.

Network attack. APRON detects network attacks including traffic manipulation. Adversaries might tamper with the network traffic between the APRON client and deployment server to deliver a manipulated system image. However, since APRON traffic is secured with TLS, the adversaries cannot arbitrarily manipulate it unless they break TLS or compromise the deployment server. Even if they succeed, APRON drops such manipulation because it verifies fetched data using the signed hash tree.

8 Discussion

In this section, we discuss some possible alternatives to APRON's design.

Advanced deduplication and compression. APRON currently uses a simple block-based deduplication (§4.4) without network traffic compression, resulting in relatively high network usage (§6.4). APRON can reduce it using advanced deduplication techniques like content-defined chunking [31,67,80] and seekable compression [116], but there are two tradeoffs. First, their computational overhead is higher than block-based deduplication, increasing the overall recovery time and runtime overhead. Second, they are incompatible with the efficient, block-level hash tree [112] APRON leverages. To maintain the compatibility, APRON requires a fine-grained hash tree with complicated data structure and computational complexity. We leave balancing these tradeoffs to future work.

File-based recovery. APRON verifies and renovates the entire storage or partition as it assumes the image-based system management (§2.2). Instead, it can focus on a set of critical files if it separately maintains their root hashes using techniques like Integrity Measurement Architecture (IMA) [62] and `fs-verity` [113]. This file-based recovery potentially reduces the overhead of APRON, but it must overcome two problems. First, it requires a separate technique to recover the kernel and filesystem itself. Otherwise, it even cannot identify whether certain files exist in the device. Second, it must maintain and update a set of root hash values in a scalable and consistent manner. This is because there are many critical files depending on each other (e.g., system binaries and shared libraries). APRON is free from such problems because it is independent from the filesystem and it only maintains a single root hash value for the entire image.

Mutable data. APRON leverages and ensures the read-only property of the image-based system management, but it does not prevent users from storing any data in the device. Like other image-based operating systems, APRON can maintain a separate read-writable user partition (§2.2). Then, APRON can let users use it as a writable overlay for the storage layer (as explained in §5) or mounting it at specific writable directories (e.g., `/etc`, `/home`) [33].

Single point of failure. APRON can suffer from a single-point-of-failure problem because its recovery task relies on a server which is remote in most cases. To overcome it, APRON needs other techniques like a load balancer [21] to mitigate this problem. Especially, the HTTPS version of APRON seamlessly benefits from such load balancing (§4.5).

9 Related Work

In this section, we explain existing studies related to APRON.

Network boot. Datacenter administrators frequently provision operating systems on new or failed server machines. They use the Preboot eXecution Environment (PXE) boot [55] to make each server boot into a small operating system stored in a storage server within the same local network. This small operating system downloads a full operating system to the local storage and, finally, boots into it. However, the PXE boot neither efficiently downloads system images nor ensures any network-level security because it relies on TFTP [106]. Thus, it can only be used within a well-managed local network. To overcome these problems, iPXE and UEFI support HTTP(S) boot [56, 59]. Still, they must download an entire operating system image to local storage to boot into it unlike APRON.

Diskless boot. Administrators can configure an operating system to use network storage as its root filesystem via remote block storage protocols (e.g., iSCSI and NBD) or network file systems (e.g., NFS and Samba). It is known as diskless boot [35,48,79,97]. It makes much more sense in a data center where servers are connected through the same high-bandwidth and low-latency local network [20]. However, since this approach fully relies on network storage, it cannot avoid repetitive fetching of the same blocks from storage servers if the blocks are evicted from the cache due to memory pressure. Further, a lack of required blocks due to potential network errors can result in significant system malfunctioning. Data block caching [26, 100, 101, 109, 110] might mitigate these problems, but cached blocks can be evicted according to the cache replacement policy unlike APRON. Also, all cached blocks should be accessed via a translation layer and discarded when any recovery or update is needed. Advanced distributed file systems [2, 101, 123] can avoid some of the problems, but they have large code bases and require complicated server- and client-side configuration. Unlike them, APRON only requires maintaining a simple file or web server.

Operating system streaming deployment. An operating system streaming deployment [28,42,43,85,111] uses both local and network storage. While serving block requests from kernel threads and other applications using network storage, the operating system streaming deployment stores downloaded blocks at the corresponding locations of local storage. These stored blocks will be used to resolve further requests to avoid repetitive downloading of the same blocks. The operating system streaming deployment also copies not-yet-downloaded blocks from network storage to local storage in the background to eventually mirror the network storage to the local storage. However, unlike APRON, existing operating system streaming deployment mechanisms neither consider secure operating system deployment nor support selective renovations of invalid blocks. Thus, it must deploy the entire operating system image from scratch if it recognizes any corruption or the operating system image has been updated.

Efficient update. Updating an operating system or its kernel with minimal downtime is heavily studied [5, 7, 16, 19, 27, 63, 90, 99, 127, 130]. A/B update [5, 7, 16] has a separate partition to download an updated system image during execution and reboot into it. Live kernel patching [19, 27, 90, 130] hot fixes the kernel without rebooting it. Since live kernel patching cannot handle complicated changes (e.g., data layout), other schemes [63, 99, 127] leverage memory snapshot and soft reboot. However, all these mechanisms work only if an operating system or underlying systems software (i.e., hypervisor [99], System Management Mode (SMM) [130]) is not compromised or corrupt. For example, a privileged attacker can tamper with both A/B partitions, hinder hot-patching, or corrupt memory or storage snapshots. Thus, they should rely on recovery mechanisms APRON to fix corrupt systems.

Multi-node progressive recovery. Multi-node progressive recovery [3, 57, 92, 129] is another way to recover or update a system with minimal or zero downtime. To maximize the availability of a critical service, these schemes operate redundant copies of the same service in multiple physical or virtual computing nodes. If the service needs to be recovered or updated, they first deal with a part of the nodes while running the other part of nodes to keep alive the service. They handle the latter part of nodes after they have recovered or updated the former part of the nodes. These schemes can achieve zero downtime if at least one node is always running. However, their resource costs are high because they require multiple nodes. In addition, we note that they can benefit from APRON because, in the end, they recover or update each node—reducing node recovery or update time is important to maintain their overall fault tolerance.

10 Conclusion

APRON is a novel approach to authentically and progressively renovate an operating system image during the system boot and after the startup of the operating system, minimizing the system downtime needed for a recovery. It is especially effective for the reboot-based security systems that frequently reset and repair devices to deal with attacks and failures, and mission-critical systems which are sensitive to the downtime. APRON renovates the entire operating system image with negligible runtime overhead and small network usage.

Acknowledgment

We would like to thank the anonymous reviewers and our shepherd, Eric Eide, for their helpful feedback.

References

[1] A Fedora initiative. Fedora Silverblue. https://silverblue.fedoraproject.org.

[2] Atul Adya, William J. Bolosky, Miguel Castro, Gerald Cermak, Ronnie Chaiken, John R. Douceur, Jon Howell, Jacob R. Lorch, Marvin Theimer, and Roger P. Wattenhofer. FARSITE: Federated, available, and reliable storage for an incompletely trusted environment. In *Proceedings of the 5th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Boston, MA, December 2002.

[3] Hussain M. J. Almohri, Layne T. Watson, and David Evans. Misery digraphs: Delaying intrusion attacks in obscure clouds. *IEEE Transactions on Information Forensics and Security*, 13(6):1361–1375, 2017.

[4] Eduardo Alvarenga, Jan R. Brands, Peter Doliwa, Jerry den Hartog, Erik Kraft, Marcel Medwed, Ventzislav Nikov, Joost Renes, Martin Rosso, Tobias Schneider, and Nikita Veshchikov. Cyber resilience for the Internet of Things: Implementations with resilience engines and attack classifications. *IEEE Transactions on Emerging Topics in Computing*, 2022.

[5] Android Open Source Project. A/B (seamless) system updates. https://source.android.com/devices/tech/ota/ab.

[6] Android Open Source Project. Implementing dm-verity. https://source.android.com/security/verifiedboot/dm-verity.

[7] Android Open Source Project. Overview of Virtual A/B. https://source.android.com/devices/tech/ota/virtual_ab.

[8] Android Open Source Project. Verified Boot. https://source.android.com/security/verifiedboot.

[9] Apple. Apple T2 security chip: Security overview, 2018. https://www.apple.com/mideast/mac/docs/Apple_T2_Security_Chip_Overview.pdf.

[10] Apple Support. About the read-only system volume in macOS Catalina. https://support.apple.com/en-us/HT210650.

[11] Apple Support. System integrity protection. https://support.apple.com/guide/security/system-integrity-protection-secb7ea06b49/1/web/1.

[12] Atakan Aral and Ivona Brandic. Dependency mining for service resilience at the edge. In *Proceedings of the 3rd IEEE/ACM Symposium on Edge Computing (SEC)*, pages 228–242, 2018.

[13] William A. Arbaugh, David J. Farber, and Jonathan M. Smith. A secure and reliable bootstrap architecture. In *Proceedings of the 18th IEEE Symposium on Security and Privacy (Oakland)*, Oakland, CA, May 1997. IEEE.

[14] Anish Athalye, Adam Belay, M. Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. Notary: A device for secure transaction approval. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*, Ontario, Canada, October 2019.

[15] Ahmed M Azab, Peng Ning, Jitesh Shah, Quan Chen, Rohan Bhutkar, Guruprasad Ganesh, Jia Ma, and Wenbo Shen. Hypervision across worlds: Real-time kernel protection from the Arm TrustZone secure world. In *Proceedings of the 21st ACM Conference on Computer and Communications Security (CCS)*, Scottsdale, Arizona, November 2014.

[16] Stefano Babic. Software management on embedded systems. https://sbabic.github.io/swupdate/overview.html.

[17] Yechan Bae, Sarbartha Banerjee, Sangho Lee, and Marcus Peinado. Spacelord: Private and secure smart space sharing. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, 2022.

[18] Adrian Baldwin, Tristan Caulfield, Marius-Constantin Ilau, and David Pym. Modelling organizational recovery. In *Proceedings of the International Conference on Simulation Tools and Techniques (SIMUtools)*, 2021.

[19] Andrew Baumann, Gernot Heiser, Jonathan Appavoo, Dilma Da Silva, Orran Krieger, Robert W. Wisniewski, and Jeremy Kerr. Providing dynamic update in an operating system. In *Proceedings of the 2005 USENIX Annual Technical Conference (ATC), General Track*, pages 279–291, 2005.

[20] Nick Black. Dynamic iSCSI at scale: Remote paging at Google, 2015. Linux Plumbers Conference.

[21] Tony Bourke. *Server Load Balancing*. " O'Reilly Media, Inc.", 2001.

[22] P. T. Breuer. The network block device, 2000. https://www.linuxjournal.com/article/3778.

[23] Neil Brown. Overlay filesystem. https://www.kernel.org/doc/Documentation/filesystems/overlayfs.txt.

[24] Kevin R. B. Butler, Stephen McLaughlin, and Patrick D. McDaniel. Rootkit-resistant disks. In *Proceedings of the 15th ACM Conference on Computer and Communications Security (CCS)*, Alexandria, VA, October 2008.

[25] Canonical Ltd. Ubuntu Core. https://ubuntu.com/core.

[26] Ramesh Chandra, Nickolai Zeldovich, Constantine Sapuntzakis, and Monica S. Lam. The Collective: A cache-based system management architecture. In *Proceedings of the 2nd USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, Boston, MA, May 2005.

[27] Yue Chen, Yulong Zhang, Zhi Wang, Liangzhao Xia, Chenfu Bao, and Tao Wei. Adaptive Android kernel live patching. In *Proceedings of the 26th USENIX Security Symposium (Security)*, Vancouver, Canada, August 2017.

[28] David Clerc, Luis Garcés-Erice, and Sean Rooney. OS streaming deployment. In *Proceedings of the International Performance Computing and Communications Conference (IPCCC)*, 2010.

[29] Eric Cole. *Advanced Persistent Threat: Understanding the Danger and How to Protect Your Organization*. Newnes, 2012.

[30] coreboot. coreboot. https://coreboot.org.

[31] Landon P Cox, Christopher D Murray, and Brian D Noble. Pastiche: Making backup cheap and easy. In *Proceedings of the 5th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Boston, MA, December 2002.

[32] Debian Manpages. nbd-client. https://manpages.debian.org/testing/nbd-client/nbd-client.8.en.html.

[33] Debian Wiki. ReadonlyRoot. https://wiki.debian.org/ReadonlyRoot.

[34] Matthew Endsley. bsdiff/bspatch. https://github.com/mendsley/bsdiff.

[35] Christian Engelmann, Hong Ong, and Stephen L. Scott. Evaluating the shared root file system approach for diskless high-performance computing systems. In *Proceedings of the 10th LCI International Conference on High-Performance Clustered Computing (HPCC)*, 2009.

[36] Fedora Docs. Fedora CoreOS. https://docs.fedoraproject.org/en-US/fedora-coreos/.

[37] Fedora Docs. Fedora Internet of Things. https://docs.fedoraproject.org/en-US/iot/.

[38] Tim Fisher. What is the Windows Boot Manager (BOOTMGR)?, 2020. https://www.lifewire.com/windows-boot-manager-bootmgr-2625813.

[39] Flatcar Project Contributors. Flatcar Container Linux. https://www.flatcar.org.

[40] Jessie Frazelle. Opening up the baseboard management controller. *Communications of the ACM*, 63(2):38–40, 2020.

[41] Free Software Foundation (FSF). GNU GRUB - GNU project. https://www.gnu.org/software/grub/.

[42] Luis Garces-Erice and Sean Rooney. Scaling OS streaming through minimizing cache redundancy. In *Proceedings of the 31st International Conference on Distributed Computing Systems Workshops (ICDCSW)*, 2011.

[43] Luis Garcés-Erice and Sean Rooney. Secure lazy provisioning of virtual desktops to a portable storage device. In *Proceedings of the 6th International Workshop on Virtualization Technologies in Distributed Computing Date (VTDC)*, 2012.

[44] Tal Garfinkel and Mendel Rosenblum. A virtual machine introspection based architecture for intrusion detection. In *Proceedings of the 10th Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2003.

[45] Google Cloud. Container-Optimized OS from Google documentation. https://cloud.google.com/container-optimized-os/docs.

[46] Google Cloud. Security overview Container-Optimized OS. https://cloud.google.com/container-optimized-os/docs/concepts/security.

[47] Vivek Goyal. kexec: A new system call to allow in kernel loading, 2014. https://lwn.net/Articles/582711/.

[48] Riccardo Gusella. The analysis of diskless workstation traffic on an Ethernet. Technical report, CALIFORNIA UNIV BERKELEY COMPUTER SYSTEMS RESEARCH GROUP, 1987.

[49] HP. The netperf homepage. https://hewlettpackard.github.io/netperf/.

[50] HP. HP Sure Recover whitepaper, 2021. https://www8.hp.com/h20195/v2/GetPDF.aspx/4AA7-4556ENW.pdf.

[51] Manuel Huber, Stefan Hristozov, Simon Ott, Vasil Sarafov, and Marcus Peinado. The lazarus effect: Healing compromised devices in the internet of small things. In *Proceedings of the 15th ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, Taipei, Taiwan, October 2020.

[52] Trammell Hudson. safeboot. https://safeboot.dev.

[53] IBM. What is destructive malware?, 2019. https://www.ibm.com/downloads/cas/XZGZLRVD.

[54] Ahmad Ibrahim, Ahmad-Reza Sadeghi, and Gene Tsudik. HEALED: Healing & attestation for low-end embedded devices. In *Proceedings of the International Conference on Financial Cryptography and Data Security (FC)*, 2019.

[55] Intel Corporation. Preboot Execution Environment (PXE) Specification Version 2.1, 1999.

[56] iPXE. iPXE - open source boot firmware. http://ipxe.org.

[57] Genya Ishigaki, Siddartha Devic, Riti Gour, and Jason P Jue. DeepPR: Progressive recovery for interdependent VNFs with deep reinforcement learning. *IEEE Journal on Selected Areas in Communications*, 38(10):2386–2399, 2020.

[58] Jeremy Cowan, Sai Charan Teja Gopaluni, and Vijay K Sikha. Security features of Bottlerocket, an open source Linux-based operating system, 2021. https://aws.amazon.com/blogs/opensource/security-features-of-bottlerocket-an-open-source-linux-based-operating-system/.

[59] Wu Jiaxin, Fu Siyuan, and Brian Richardson. Getting started with UEFI HTTPS boot on EDK II. https://laurie0131.gitbooks.io/getting-started-with-uefi-https-boot-on-edk-ii/content/.

[60] Richard W. M. Jones. nbdkit. https://gitlab.com/nbdkit/nbdkit.

[61] Samuel Karp. Bottlerocket: A special-purpose container operating system, 2020. https://aws.amazon.com/blogs/containers/bottlerocket-a-special-purpose-container-operating-system/.

[62] Dmitry Kasatkin, David Safford, and Mimi Zohar. Integrity measurement architecture (IMA) wiki. https://sourceforge.net/p/linux-ima/wiki/Home/.

[63] Sanidhya Kashyap, Changwoo Min, Byoungyoung Lee, Taesoo Kim, and Pavel Emelyanov. Instant OS updates via userspace checkpoint-and-restart. In *Proceedings of the 2016 USENIX Annual Technical Conference (ATC)*, Denver, CO, June 2016.

[64] Aaron Kili. systemd-analyze – find system boot-up performance statistics in Linux, 2018. https://www.tecmint.com/systemd-analyze-monitor-linux-bootup-performance/.

[65] Jan Kneschke. lighttpd. https://www.lighttpd.net.

[66] Xeno Kovah and Corely Kallenberg. Advanced x86: BIOS and system management mode internals SPI flash protection mechanisms, 2014. http://opensecuritytraining.info/IntroBIOS.html.

[67] Erik Kruus, Cristian Ungureanu, and Cezary Dubnicki. Bimodal content defined chunking for backup streams. In *Proceedings of the 8th USENIX Conference on File and Storage Technologies (FAST)*, San Jose, CA, February 2010.

[68] Eva-Katharina Kunst and Jürgen Quade. Linux control over secure boot, 2018. https://www.linux-magazine.com/Issues/2018/206/Linux-Secure-Boot-with-Shim.

[69] Hojoon Lee, Hyungon Moon, Daehee Jang, Kihwan Kim, Jihoon Lee, Yunheung Paek, and Brent ByungHoon Kang. Ki-Mon: A hardware-assisted event-triggered monitoring platform for mutable kernel object. In *Proceedings of the 22th USENIX Security Symposium (Security)*, Washington, DC, August 2013.

[70] Joshua MacDonald. Xdelta. https://github.com/jmacd/xdelta.

[71] Larry McVoy and Carl Staelin. lmbench: Portable tools for performance analysis. In *Proceedings of the 1996 USENIX Annual Technical Conference (ATC)*, January 1996.

[72] Ralph C Merkle. Protocols for public key cryptosystems. In *Proceedings of the 1980 IEEE Symposium on Security and Privacy (Oakland)*, Oakland, CA, April 1980.

[73] Justin Meza, Qiang Wu, Sanjev Kumar, and Onur Mutlu. A large-scale study of flash memory failures in the field. In *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, 2015.

[74] Microsoft Docs. State separation and isolation. https://docs.microsoft.com/en-us/hololens/security-state-separation-isolation.

[75] Microsoft Docs. Windows recovery environment (Windows RE), 2017. https://docs.microsoft.com/en-us/windows-hardware/manufacture/desktop/windows-recovery-environment--windows-re--technical-reference.

[76] Microsoft Docs. Kernel soft reboot in Azure Stack HCI, 2021. https://docs.microsoft.com/en-us/azure-stack/hci/manage/kernel-soft-reboot.

[77] Microsoft Threat Intelligence Center. Destructive malware targeting Ukrainian organizations, 2022. https://www.microsoft.com/security/blog/2022/01/15/destructive-malware-targeting-ukrainian-organizations/.

[78] Hyungon Moon, Hojoon Lee, Jihoon Lee, Kihwan Kim, Yunheung Paek, and Brent Byunghoon Kang. Vigilare: Toward snoop-based kernel integrity monitor. In *Proceedings of the 19th ACM Conference on Computer and Communications Security (CCS)*, Raleigh, NC, October 2012.

[79] Amin Mosayyebzadeh, Apoorve Mohan, Sahil Tikale, Mania Abdi, Nabil Schear, Charles Munson, Trammell Hudson, Larry Rudolph, Gene Cooperman, Peter Desnoyers, and Orran Krieger. Supporting security sensitive tenants in a bare-metal cloud. In *Proceedings of the 2019 USENIX Annual Technical Conference (ATC)*, Renton, WA, July 2019.

[80] Athicha Muthitacharoen, Benjie Chen, and David Mazieres. A low-bandwidth network file system. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP)*, Chateau Lake Louise, Banff, Canada, October 2001.

[81] Ramin Nafisi. FoggyWeb: Targeted NOBELIUM malware leads to persistent backdoor, 2021. https://www.microsoft.com/security/blog/2021/09/27/foggyweb-targeted-nobelium-malware-leads-to-persistent-backdoor/.

[82] National Cybersecurity and Communications Integration Center. Destructive malware, 2017. https://www.cisa.gov/uscert/sites/default/files/documents/Destructive_Malware_White_Paper_S508C.pdf.

[83] NixOS contributors. Nix: Reproducible builds and deployments. https://nixos.org.

[84] Shadi A Noghabi, Landon Cox, Sharad Agarwal, and Ganesh Ananthanarayanan. The emerging landscape of edge computing. *GetMobile: Mobile Computing and Communications*, 23(4):11–20, 2020.

[85] Yushi Omote, Takahiro Shinagawa, and Kazuhiko Kato. Improving agility and elasticity in bare-metal clouds. In *Proceedings of the 20th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Istanbul, Turkey, March 2015.

[86] openSUSE contributors. openSUSE MicroOS. https://microos.opensuse.org.

[87] Perception Point. Technical analysis of CVE-2022-22583: Bypassing macOS system integrity protection (SIP), 2022. https://perception-point.io/technical-analysis-of-cve-2022-22583-bypassing-macos-system-integrity-protection/.

[88] Nick L Petroni Jr, Timothy Fraser, Jesus Molina, and William A Arbaugh. Copilot-a coprocessor-based kernel runtime integrity monitor. In *Proceedings of the 13rd USENIX Security Symposium (Security)*, San Diego, CA, August 2004.

[89] Phoronix Media. Phoronix Test Suite - Linux testing & benchmarking platform, automated testing, open-source benchmarking. https://www.phoronix-test-suite.com.

[90] Josh Poimboeuf. Introducing kpatch: Dynamic kernel patching, 2014. https://www.redhat.com/en/blog/introducing-kpatch-dynamic-kernel-patching.

[91] Martin Pool. rdiff(1) - Linux man page. https://linux.die.net/man/1/rdiff.

[92] Mahsa Pourvali, Kaile Liang, Feng Gu, Hao Bai, Khaled Shaban, Samee Khan, and Nasir Ghani. Progressive recovery for network virtualization after large-scale disasters. In *Proceedings of the 2016 International Conference on Computing, Networking and Communications (ICNC)*, 2016.

[93] Red Hat. Fedora Cloud. https://alt.fedoraproject.org/cloud/.

[94] Red Hat Customer Portal. 32.2. Anaconda rescue mode Red Hat Enterprise Linux 7. https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/7/html/installation_guide/sect-rescue-mode.

[95] Red Hat Customer Portal. Appendix A. The Device Mapper Red Hat Enterprise Linux 7. https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/7/html/logical_volume_manager_administration/device_mapper.

[96] Red Hat Software. Red Hat Enterprise Linux Atomic Host: A platform optimized for Linux containers.

[97] Paul Riddle. Automated upgrades in a lab environment. In *Proceedings of the 8th USENIX Symposium on Large Installation System Administration Conference (LISA)*, 1994.

[98] Jonas Röckl, Mykolai Protsenko, Monika Huber, Tilo Müller, and Felix C Freiling. Advanced system resiliency based on virtualization techniques for IoT devices. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, 2021.

[99] Mark Russinovich, Naga Govindaraju, Melur Raghuraman, David Hepkin, Jamie Schwartz, and Arun Kishan. Virtual machine preserving host updates for zero day patching in public cloud. In *Proceedings of the 16th European Conference on Computer Systems (EuroSys)*, Virtual, April 2021.

[100] Constantine P. Sapuntzakis, Ramesh Chandra, Ben Pfaff, Jim Chow, Monica S. Lam, and Mendel Rosenblum. Optimizing the migration of virtual computers. In *Proceedings of the 5th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Boston, MA, December 2002.

[101] Mahadev Satyanarayanan, James J. Kistler, Puneet Kumar, Maria E. Okasaki, Ellen H. Siegel, and David C. Steere. Coda: A highly available file system for a distributed workstation environment. *IEEE Transactions on Computers*, 39(4):447–459, 1990.

[102] Uday Savagaonkar, Nelly Porter, Nadim Taha, Benjamin Serebrin, and Neal Mueller. Titan in depth: Security in plaintext, 2017. https://cloud.google.com/blog/products/gcp/titan-in-depth-security-in-plaintext.

[103] Bianca Schroeder, Raghav Lagisetty, and Arif Merchant. Flash reliability in production: The expected and the unexpected. In *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST)*, Santa Clara, CA, February 2016.

[104] Quentin Schulz and Mylène Josserand. Secure boot from A to Z, 2018. Embedded Linux Conference.

[105] Sidero Labs, Inc. Talos Linux. https://www.talos.dev.

[106] K Sollins. The TFTP protocol (revision 2). Technical report, STD 33, RFC 1350, MIT, 1992.

[107] Daniel Stenberg. curl. https://curl.se.

[108] Kuniyasu Suzaki, Akira Tsukamoto, Andy Green, and Mohammad Mannan. Reboot-oriented IoT: Life cycle management in trusted execution environment for disposable IoT devices. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, 2020.

[109] The kernel development community. A block layer cache (bcache). https://www.kernel.org/doc/html/latest/admin-guide/bcache.html.

[110] The kernel development community. dm-cache. https://www.kernel.org/doc/html/latest/admin-guide/device-mapper/cache.html.

[111] The kernel development community. dm-clone. https://www.kernel.org/doc/html/latest/admin-guide/device-mapper/dm-clone.html.

[112] The kernel development community. dm-verity. https://www.kernel.org/doc/html/latest/admin-guide/device-mapper/verity.html.

[113] The kernel development community. fs-verity: read-only file-based authenticity protection. https://www.kernel.org/doc/html/latest/filesystems/fsverity.html.

[114] The kernel development community. kcopyd. https://www.kernel.org/doc/html/latest/admin-guide/device-mapper/kcopyd.html.

[115] The OpenSSL Project Authors. OpenSSL – cryptography and SSL/TLS toolkit. https://openssl.org.

[116] The Tukaani Project. XZ Utils. https://tukaani.org/xz/.

[117] Josh Triplett. Chrome OS internals, 2014. LinuxCon Europe.

[118] Trusted Computing Group. TPM 2.0 Authenticated Countdown Timer (ACT) Command, 2019.

[119] Trusted Computing Group. Trusted Platform Module Library - Part 1: Architecture, 2019.

[120] Ubuntu documentation. LiveCdRecovery. https://help.ubuntu.com/community/LiveCdRecovery.

[121] UEFI Forum. Unified Extensible Firmware Interface (UEFI) specification, version 2.9, 2021.

[122] Kushagra Vaid. Microsoft creates industry standards for datacenter hardware storage and security, 2018. https://azure.microsoft.com/en-us/blog/microsoft-creates-industry-standards-for-datacenter-hardware-storage-and-security/.

[123] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Seattle, WA, November 2006.

[124] Richard Wilkins and Brian Richardson. UEFI secure boot in modern computer security solutions, 2013.

[125] Meng Xu, Manuel Huber, Zhichuang Sun, Paul England, Marcus Peinado, Sangho Lee, Andrey Marochko, Dennis Mattoon, Rob Spiger, and Stefan Thom. Dominance as a new trusted computing primitive for the Internet of Things. In *Proceedings of the 40th IEEE*

Symposium on Security and Privacy (Oakland), San Francisco, CA, May 2019.

[126] Ron Yorston. Keeping filesystem images sparse. https://frippery.org/uml/.

[127] Anthony Yznaga. PKRAM: Preserved-over-Kexec RAM. https://lwn.net/Articles/851192/.

[128] Fengwei Zhang, Kevin Leach, Kun Sun, and Angelos Stavrou. SPECTRE: A dependable introspection framework via System Management Mode. In *Proceedings of the 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2013.

[129] Yangming Zhao, Mohammed Pithapur, and Chunming Qiao. On progressive recovery in interdependent cyber physical systems. In *Proceedings of the 2016 IEEE Global Communications Conference (GLOBECOM)*, 2016.

[130] Lei Zhou, Fengwei Zhang, Jinghui Liao, Zhengyu Ning, Jidong Xiao, Kevin Leach, Westley Weimer, and Guojun Wang. KShot: Live kernel patching with SMM and SGX. In *Proceedings of the 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2020.

A Merkle Hash Tree

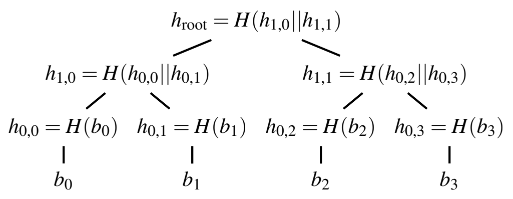

$h_{\text{root}} = H(h_{1,0}||h_{1,1})$
$h_{1,0} = H(h_{0,0}||h_{0,1})$
$h_{1,1} = H(h_{0,2}||h_{0,3})$
$h_{0,0} = H(b_0)$
$h_{0,1} = H(b_1)$
$h_{0,2} = H(b_2)$
$h_{0,3} = H(b_3)$
b_0
b_1
b_2
b_3

Figure 11: Merkle hash tree

A Merkle hash tree [72] is a method to efficiently and securely verify whether any part of data is corrupt. It is constructed by recursively computing hashes over data and their hashes (Figure 11). Its root hash summarizes the entire data. Thus, we only need to ensure the root hash's authenticity and integrity (i.e., sign it) to verify data and node hashes. For example, to verify a data block b'_1, we compute $h'_{0,1} = H(b'_1)$, $h'_{1,0} = H(h_{0,0}||h'_{0,1})$, and $h'_{\text{root}} = H(h'_{1,0}||h_{1,1})$ with leaf and internal node hashes $h_{0,0}$ and $h_{1,1}$—which have been verified in the same manner—and compare h'_{root} with signed h_{root}.

B Delta Update

The detailed procedure of the delta update is below. First, we compute an `rdiff` signature which is a structured summary of a base file (i.e., a corrupt system partition) to compute delta. In our APRON device, it takes ∼21 s to compute an `rdiff` signature over the system partition regardless of how many portions of it are corrupt. The signature size is 181 MiB without compression. Once we compress it with `gzip`, it becomes between 72 MiB (1% corruption) and 0.5 MiB (100% corruption). Next, we upload the compressed signature to the server, decompress it, and compute the delta between the signature and the valid system image. The delta computation takes 55–273 s and the size of the delta is between 91 MiB and 4.1 GiB (between 31 MiB and 1.6 GiB after compression). Both depend on how many portions of the system partition are corrupt. We only use the Azure VM for this delta computation to ignore the CPU performance difference between the two servers. Finally, we download the compressed delta—which take 1–16 s (high throughput) and 9–135 s (low throughput), decompress it, and patch the system partition with it. Patching itself takes ∼12 s regardless of the number of corrupt blocks.