# StrongBox: Confidentiality, Integrity, and Performance using Stream Ciphers for Full Drive Encryption

Bernard Dickens III
University of Chicago
bd3@cs.uchicago.edu

Haryadi S. Gunawi
University of Chicago
haryadi@cs.uchicago.edu

Ariel J. Feldman
University of Chicago
arielfeldman@cs.uchicago.edu

Henry Hoffmann
University of Chicago
hankhoffmann@cs.uchicago.edu

## Abstract

Full-drive encryption (FDE) is especially important for mobile devices because they contain large quantities of sensitive data yet are easily lost or stolen. Unfortunately, the standard approach to FDE—the AES block cipher in XTS mode—is 3–5x slower than unencrypted storage. Authenticated encryption based on stream ciphers is already used as a faster alternative to AES in other contexts, such as HTTPS, but the conventional wisdom is that stream ciphers are unsuitable for FDE. Used naively in drive encryption, stream ciphers are vulnerable to attacks, and mitigating these attacks with on-drive metadata is generally believed to ruin performance.

In this paper, we argue that recent developments in mobile hardware invalidate this assumption, making it possible to use fast stream ciphers for FDE. Modern mobile devices employ solid-state storage with Flash Translation Layers (FTL), which operate similarly to Log-structured File Systems (LFS). They also include trusted hardware such as Trusted Execution Environments (TEEs) and secure storage areas. Leveraging these two trends, we propose StrongBox, a stream cipher-based FDE layer that is a drop-in replacement for dm-crypt, the standard Linux FDE module based on AES-XTS. StrongBox introduces a system design and on-drive data structures that exploit LFS's lack of overwrites to avoid costly rekeying and a counter stored in trusted hardware to protect against attacks. We implement StrongBox on an ARM big.LITTLE mobile processor and test its performance under multiple popular production LFSes. We find that StrongBox improves read performance by as much as 2.36× (1.72× on average) while offering stronger integrity guarantees.

***CCS Concepts*** • **Information systems → Data encryption**; *Flash memory*; • **Security and privacy → Block and stream ciphers**; **Hash functions and message authentication codes**; *Key management*; Tamper-proof and tamper-resistant designs; • **Software and its engineering → File systems management**;

***Keywords*** AES-XTS; full disk encryption; replay protected memory block; log-structured; high read performance; dm-crypt

**ACM Reference Format:**
Bernard Dickens III, Haryadi S. Gunawi, Ariel J. Feldman, and Henry Hoffmann. 2018. StrongBox: Confidentiality, Integrity, and Performance using Stream Ciphers for Full Drive Encryption. In *ASPLOS '18: 2018 Architectural Support for Programming Languages and Operating Systems, March 24–28, 2018, Williamsburg, VA, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3173162.3173183


*ASPLOS'18, March 24–28, 2018, Williamsburg, VA, USA*

ACM ISBN 978-1-4503-4911-6/18/03…$15.00
https://doi.org/10.1145/3173162.3173183

## 1 Introduction

Full-drive encryption (FDE)[1] is an essential technique for protecting the privacy of data at rest. For mobile devices, maintaining data privacy is especially important as these devices contain sensitive personal and financial data yet are easily lost or stolen. The current standard for securing data at rest is to use the AES cipher in XTS mode [4, 5]. Unfortunately, employing AES-XTS increases read/write latency by 3–5× compared to unencrypted storage.

It is well known that authenticated encryption using *stream* ciphers—such as ChaCha20 [12]—is faster than using AES (see Fig. 1). Indeed, Google made the case for stream ciphers over AES, switching HTTPS connections on Chrome for Android to use a stream cipher for better performance [9]. Stream ciphers are not used for FDE, however, for two reasons: (1) confidentiality and (2) performance. First, when applied naively to stored data, stream ciphers are trivially

[1]The common term is full-*disk* encryption, but this work targets SSDs, so we use *drive*.

vulnerable to attacks—including *many-time pad and rollback attacks*—that reveal the plaintext by overwriting a secure storage location with the same key. Second, it has been assumed that adding the meta-data required to resist these attacks would ruin the stream cipher's performance advantage. Thus, the conventional wisdom is that FDE necessarily incurs the overhead of AES-XTS or a similar primitive.

We argue that two technological shifts in mobile device hardware overturn this conventional wisdom, enabling confidential, high-performance storage with stream ciphers. First, these devices commonly employ solid-state storage with Flash Translation Layers (FTL), which operate similarly to Log-structured File Systems (LFS) [21, 22, 27]. Second, mobile devices now support trusted hardware, such as Trusted Execution Environments (TEE) [24, 29] and secure storage areas [10]. FTLs and LFSes are used to limit sector/cell overwrites, hence extending the life of the drive. Most writes simply appended to a log, reducing the occurrence of overwrites and the chance for attacks. The presence of secure hardware means that drive encryption modules have access to persistent, monotonically increasing counters that can be used to prevent rollback attacks when overwrites do occur.

Given these trends, we propose StrongBox, a new method for securing data at rest. StrongBox is a drop-in replacement for AES-XTS-backed FDE such as dm-crypt [8]; *i.e.,* it requires no interface changes. The primary challenge is that even with a FTL or LFS running above an SSD, filesystem blocks will occasionally be overwritten; *e.g.,* by segment cleaning or *garbage collection.* StrongBox overcomes this challenge by using a fast stream cipher for confidentiality and performance with integrity preserving Message Authentication Codes [6] or "MAC tags" and a secure, persistent hardware counter to ensure integrity and prevent attacks. *StrongBox's main contribution is a system design enabling the first confidential, high- performance drive encryption based on a stream cipher.*

We demonstrate StrongBox's effectiveness on a mobile ARM big.LITTLE system—a Samsung Exynos Octa 5—running Ubuntu Trusty 14.04 LTS, kernel 3.10.58. We use ChaCha20 [12] as our stream cipher, Poly1305 [11] as our MAC algorithm, and the eMMC Replay Protected Memory Block partition to store a secure counter [10]. As StrongBox requires no change to any existing interfaces, we benchmark it on two of the most popular LFSes: NILFS [21] and F2FS [22]. We compare the performance of these LFSes on top of AES-XTS (via dm-crypt) and StrongBox. Additionally, we compare the performance of AES-XTS encrypted Ext4 filesystems with StrongBox and F2FS. Our results show:

- *Improved read performance:* StrongBox provides decreased read latencies across all tested filesystems in the majority of benchmarks when compared to dm-crypt; *e.g.,* under F2FS, StrongBox provides as much as a 2.36× (1.72× average) speedup over AES-XTS.

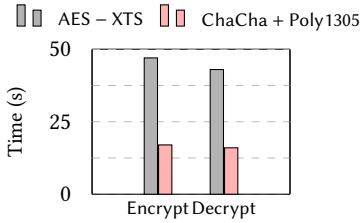


**Figure 1.** AES-XTS and ChaCha20+Poly1305 Comparison.

- *Equivalent write performance:* despite having to maintain more metadata than FDE schemes based on AES-XTS, StrongBox achieves near parity or provides an improvement in observed write latencies in the majority of benchmarks; *e.g.,* under F2FS, StrongBox provides an average 1.27× speedup over AES-XTS.

StrongBox achieves these performance gains while providing a stronger integrity guarantee than AES-XTS. Whereas XTS mode only hopes to randomize plaintext when the ciphertext is altered [4], StrongBox provides the security of standard authenticated encryption. In addition, StrongBox's implementation is available open-source.[2]

## 2 Motivation

We detail the main motivations for StrongBox: stream ciphers' speed compared to AES-XTS and Log-structured File Systems' append-mostly nature. We then describe the challenges of replacing AES with a stream cipher.

### 2.1 Performance Potential

We demonstrate the potential performance win from switching to a stream cipher by comparing AES-XTS to ChaCha20+Poly1305. We use an Exynos Octa processor with an ARM big.LITTLE architecture—the same processor used in the Samsung Galaxy line of phones. We encrypt and then decrypt 250MB of randomly generated bits 3 times and take the median time for each of encryption and decryption. Fig. 1 shows the distinct advantage of the stream cipher over AES—a consistent 2.7× reduction in run time.

### 2.2 Append-mostly Filesystems

Of course, stream ciphers are not designed to encrypt data at rest. If we naively implement block device encryption with a stream cipher, overwriting the same memory location with the same key would trivially allow an attacker to recover the secret key. Thus we believe stream ciphers are best suited for encrypting block devices backing Log-structured File Systems (LFSes), as these filesystems are designed to append data to the end of a log rather than overwrite data. In practice, some overwrites occur; *e.g.,* in metadata, but they are small in number during normal execution.

[2] https://git.xunn.io/research/buselfs-public

To demonstrate this fact, we write 800MB of random data directly to the backing store using four different file systems: Ext4, LogFS, NILFS, and F2FS. We count the number of total writes to the underlying block device and the number of times data is overwritten for each file system.

**Table 1.** File System Overwrite Behavior

| File System | Total Write Ops | Overwrites |
|---|---|---|
| ext4 | 16,756 | 10,787 |
| LogFS | 4,244 | 32 |
| NILFS | 4,199 | 24 |
| F2FS | 2.107 | 2 |

Table 1 shows the data for this experiment. Ext4 has the highest number of writes, but many of those are small writes for book-keeping purposes. Ext4 also has the largest number of overwrites. Almost 65% of the writes are to a previously written location in the backing store. In contrast, all three Log-structured file systems have very few overwrites.

### 2.3 Threat Model

A stream cipher can be more than twice as fast as AES-XTS while providing the same confidentiality guarantee. The problem is that a stream cipher is not secure if the same key is used to overwrite the same storage location. Fortunately, FTLs and LFSes rarely overwrite the same location.

We cannot, however, ignore the fact that overwrites do occur. While Table 1 shows overwrites are rare during normal operation, we know they will occur when garbage collecting the LFS. Thus, we will need some metadata to track writes and ensure that data is handled securely if overwrites occur. Therefore, we recognize three key challenges to replacing AES with a stream cipher for FDE:

- Tracking writes to the block device to ensure that the same location is never overwritten with the same key.
- Ensuring that the metadata that tracks writes is secure and is not subject to leaks or rollback attacks.
- Accomplishing the above efficiently so that we maintain the performance advantage of the stream cipher.

The key to StrongBox is using a secure, persistent counter supported in modern mobile hardware; *e.g.,* for limiting password attempts. This counter can track writes, and thus *versions* of the encrypted data. If an attacker tried to *roll back* the file system to overwrite the same location with the same key, our StrongBox detects that the local version number is out of sync with the global version number stored in the secure counter. In that case, StrongBox refuses to initialize and the attack fails. The use of the hardware-supported secure counter significantly raises the bar when it comes to rollback attacks, requiring a costly and non-discrete physical attack on the hardware itself to be effective. The actual structure of the metadata required to track writes and maintain integrity is significantly more complicated than simply implementing a counter and is the subject of the next section.

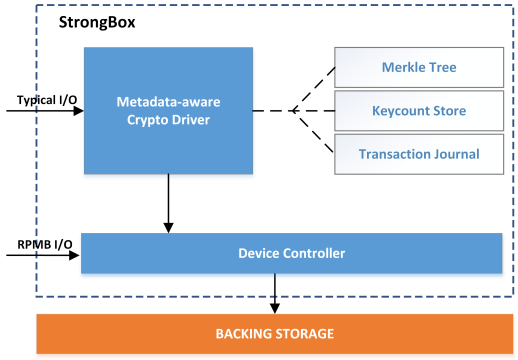


**Figure 2.** Overview of the StrongBox construction.

An additional challenge is that of crash recovery. StrongBox relies on the overlying filesystem to manage data recovery in the event of a crash that leaves user data in an inconsistent state. StrongBox handles metadata recovery after a crash by giving the root user the option to accept the current metadata state as the new consistent state, *i.e.,* "force mounting". This allows the root user to mount the filesystem and access data after an unexpected shutdown. An attacker might try to take advantage of this feature by modifying the backing store, forcing an inconsistent state, and hoping the root user will ignore it and force mount the system anyway. StrongBox defends against this attack by preventing force mounts when metadata state is wildly inconsistent with the global version counter. Otherwise, the root user is warned if they attempt a force mount. Thus, attacking StrongBox by forcing a crash can only be successful if the attacker also has root permission, in which case security is already compromised. Crash recovery is also detailed in the next section.

## 3 StrongBox System Design

StrongBox acts as a translation layer sitting between the drive and the operating system. It provides confidentiality and integrity guarantees while minimizing performance loss due to metadata management overhead. StrongBox accomplishes this by leveraging the speed of stream ciphers over the AES block cipher and taking advantage of the append-mostly nature of Log-structured Filesystems (LFS) and modern Flash Translation Layers (FTL) [14].

Hence, there are several locations where StrongBox could be implemented in the system stack. StrongBox could be integrated into an LFS kernel module itself—*e.g.,* F2FS—specifically leveraging the flexibility of the Virtual Filesystem Switch (VFS). StrongBox could be implemented as an actual block device, or virtual block device layered atop a physical block device; the latter is where we chose to implement our prototype. StrongBox could even be implemented within the SSD

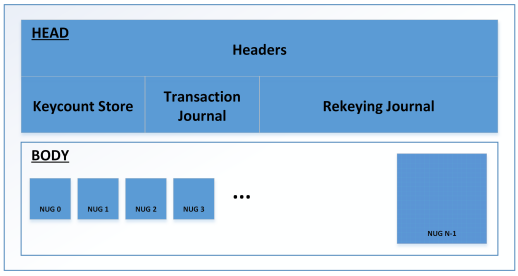


**Figure 3.** Layout of StrongBox's backing storage.

drive controller's FTL, which handles scatter gather, garbage collection, wear-leveling, etc.

Fig. 2 illustrates StrongBox's design. StrongBox's metadata is encapsulated in four components: an in-memory *Merkle Tree* and two drive-backed byte arrays—the *Keycount Store* and the *Transaction Journal*—and a persistent monotonic counter we implement with the *Replay Protected Memory Block* or RPMB. All four are integrated into the *Cryptographic Driver*, which handles data encryption, verification, and decryption during interactions with the underlying backing store. These interactions take place while fulfilling high-level I/O requests received from the LFS. The *Device Controller* handles low-level I/O between StrongBox and the backing store.

The rest of this section describes the components referenced in Fig. 2. Specifically: we first describe the backing store and StrongBox's layout for data and metadata. This is followed by an exploration of the cryptographic driver and how it interacts with that metadata, the role of the device controller, an overview of rekeying in the backing store, and further considerations to ensure confidentiality in the case of rollbacks and related attacks.

### 3.1 Backing Store Function and Layout

The backing store is the storage media on which StrongBox operates. Fig. 3 illustrates StrongBox's layout on this store.

In the *body* section of the backing store layout, end-user data is partitioned into a series of same-size *logical blocks*. These are distinct from the concept of *physical drive blocks*, which are collections of one or more drive sectors. To make this distinction clear, we refer to these wider logical blocks as *nuggets*, marked *NUG* in the Body section of Fig. 3. Hence, a nugget consists of one or more physical drive blocks, depending on its configured size. Each nugget is subdivided into a constant number of sub-blocks we refer to as *flakes*.

The reason for these nugget/flake divisions are two-fold:

1. To track, detect, and handle overwrites and
2. To limit the maximum length of any plaintexts operated on by the cryptographic driver, decreasing the overhead incurred per I/O operation and per overwrite.

Considering the first item, we are required to keep track of writes so that we may detect when an overwrite occurs. Flakes are key to this tracking. When a request comes in to write to one or more flakes in a nugget, StrongBox marks the affected flakes as "flagged". Here, "flagged" implies that another write to some portion of that flake would constitute an overwrite. If a new request comes in to write to one or more of those same flakes another time, StrongBox triggers a "rekeying" procedure over the entire nugget to safely overwrite the old data in those flakes. This rekeying procedure is necessarily time consuming, ballooning the overhead of overwrites translated by StrongBox.

Considering the second item, nugget size here governs the granularity of rekeying while flake size governs granularity when identifying overwrites. A larger nugget size will increase the penalty incurred with rekeying (you're re-encrypting a larger number of bytes) while a smaller nugget size will increase the quantity of nuggets needing to be rekeyed when an overwrite is detected as well as increase the amount of metadata stored on drive and in memory. On the other hand, a larger flake size will increase the number of times an incoming write is seen as an overwrite, with a non-optimal nugget-sized flake requiring a rekeying on *every write*. A smaller flake size will increase the amount of metadata stored on drive and in memory.

The size and structure of that metadata is described in greater detail throughout the rest of this section.

The *head* section of the backing store layout contains the metadata written to drive during StrongBox's initialization. These headers govern StrongBox's operation and are, in order:

1. VERSION, 4 bytes; specifies the StrongBox version originally used to initialize the backing store
2. SALT, 16 bytes; the salt used in part to derive the global master secret
3. MTRH, 32 bytes; the hash of the Merkle Tree root
4. TPMGLOBALVER, 8 bytes; the monotonic global version count, parity in hardware-supported secure storage
5. VERIFICATION, 32 bytes; used to determine if the key derived from a password is correct
6. NUMNUGGETS, 4 bytes; the number of nuggets contained by the backing store
7. FLAKESPERNUGGET, 4 bytes; the number of flakes/nugget
8. FLAKESIZE, 4 bytes; the size of each flake, in bytes
9. INITIALIZED, 1 byte; used to determine if the backing store has been properly initialized
10. REKEYING, 4 bytes; the index of the nugget in need of rekeying if there is a pending rekeying procedure

After the headers, two byte arrays are stored in the Head section: one of $N$ 8-byte integer *keycounts* and one of $N$ $\lceil P/8 \rceil$-byte *transaction journal entries*, where $N$ is the number of nuggets and $P$ is the number of flakes per nugget.

Finally, the *Rekeying Journal* is stored at the end of the Head section. The rekeying journal is where nuggets and

their associated metadata are transiently written, enabling StrongBox to resume rekeying in the event that it is interrupted during the rekeying procedure.

### 3.2 Metadata-aware Cryptographic Driver

The cryptographic driver coordinates StrongBox's disparate components. Its primary function is to map incoming reads and writes to their proper destinations in the backing store, applying our chosen stream cipher and message authentication code to encrypt, verify, and decrypt data on the fly with consideration for metadata management.

When a read request is received, it is first partitioned into affected nuggets; *e.g.,* a read that spans two nuggets is partitioned in half. For each nugget affected, we calculate which flakes are touched by the request. We then verify the contents of those flakes. If all the flakes are valid, whatever subset of data that was requested by the user is decrypted and returned. Algorithm 1 details StrongBox's read operation.

Like reads, when a write request is received, the request is first partitioned with respect to affected nuggets. For each affected nugget, we calculate which flakes are touched by the request and then check if any of those flakes are marked as flagged in the transaction journal. If one or more of them have been marked flagged, we trigger rekeying for this specific nugget (see: Algorithm 3) and end there. Otherwise, we mark these touched flakes as flagged in the transaction journal. We then iterate over these touched flakes. For the first and last flakes touched by the write request, we execute an internal read request (see: Algorithm 1) to both obtain the flake data and verify that data with the Merkle Tree. We then overwrite every touched flake with the data from the requested operation, update the Merkle Tree to reflect this change, encrypt and write out the new flake data, and commit all corresponding metadata. Algorithm 2 details StrongBox's write operation.

#### 3.2.1 Transaction Journal

An overwrite breaks the security guarantee offered by any stream cipher. To prevent this failure, StrongBox tracks incoming write requests to prevent overwrites. This tracking is done with the transaction journal, featured in Fig. 2.

The transaction journal consists of $N$ $\lceil P/8 \rceil$-byte bit vectors where $N$ is the number of nuggets and $P$ is the number of flakes per nugget. A bit vector $v$ contains at least $P$ bits $v = \{b_0, b_1, b_2, \ldots, b_{P-1}, \ldots\}$, with extra bits ignored. Each vector is associated with a nugget and each bit with a flake belonging to that nugget. When an incoming write request occurs, the corresponding bit vector is updated (set to 1) to reflect the new flagged state of those flakes.

The transaction journal is referenced during each write request, where it is updated to reflect the state of the nugget and checked to ensure the operation does not constitute an overwrite. If the operation *does* constitute an overwrite, StrongBox triggers a rekeying procedure for the entire nugget before safely completing the request.

**Algorithm 1** StrongBox handling an incoming read request.

**Require:** The read request is over a contiguous segment of the backing store
**Require:** $\ell, \ell' \leftarrow$ read requested length
**Require:** $\aleph \leftarrow$ master secret
**Require:** $n_{index} \leftarrow$ first nugget index to be read
$data \leftarrow empty$
**while** $\ell \neq 0$ **do**
  $k_{n_{index}} \leftarrow GenKey_{nugget}(n_{index}, \aleph)$
  Fetch nugget keycount $n_{kc}$ from Keycount Store.
  Calculate indices touched by request: $f_{first}, f_{last}$
  $n_{flakedat} \leftarrow ReadFlakes(f_{first}, \ldots, f_{last})$
  **for** $f_{current} = f_{first}$ **to** $f_{last}$ **do**
    $k_{f_{current}} \leftarrow GenKey_{flake}(k_{n_{index}}, f_{current}, n_{kc})$
    $tag_{f_{current}} \leftarrow GenMac(k_{f_{current}}, n_{flakedat}[f_{current}])$
    Verify $tag_{f_{current}}$ in Merkle Tree.
  *(*) denotes requested subset of nugget data*
  $data \leftarrow data + Decrypt(*n_{flakedat}, k_{n_{index}}, n_{kc})$
  $\ell \leftarrow \ell - \| * n_{flakedat} \|$
  $n_{index} \leftarrow n_{index} + 1$
**return** $data$ ▷ Fulfill the read request
**Ensure:** $\|data\| <= \ell'$
**Ensure:** $\ell = 0$

**Algorithm 2** StrongBox handling an incoming write request.

**Require:** The write request is to a contiguous segment of the backing store
**Require:** $\ell, \ell' \leftarrow$ write requested length
**Require:** $\aleph \leftarrow$ master secret
**Require:** $data \leftarrow$ cleartext data to be written
**Require:** $n_{index} \leftarrow$ first nugget index to be affected
Increment secure counter: by 2 if we recovered from a crash, else 1
**while** $\ell \neq 0$ **do**
  Calculate indices touched by request: $f_{first}, f_{last}$
  **if** Transaction Journal entries for $f_{first}, \ldots, f_{last} \neq 0$ **then**
    Trigger rekeying procedure (see: Algorithm 3).
    **continue**
  Set Transaction Journal entries for $f_{first}, \ldots, f_{last}$ to 1
  $k_{n_{index}} \leftarrow GenKey_{nugget}(n_{index}, \aleph)$
  Fetch nugget keycount $n_{kc}$ from Keycount Store.
  **for** $f_{current} = f_{first}$ **to** $f_{last}$ **do**
    $n_{flakedat} \leftarrow empty$
    **if** $f_{current} == f_{first} \| f_{current} == f_{last}$ **then**
      $n_{flakedat} \leftarrow CryptedRead(FSIZE, \aleph, n_{index}@f_{offset})$
    $n_{flakedat} \leftarrow Encrypt(n_{flakedat}, k_{n_{index}}, n_{kc})$
    $k_{f_{current}} \leftarrow GenKey_{flake}(k_{n_{index}}, f_{current}, n_{kc})$
    $tag_{f_{current}} \leftarrow GenMac(k_{f_{current}}, n_{flakedat})$
    Update new $tag_{f_{current}}$ in Merkle Tree.
    $WriteFlake(f_{current}, n_{flakedat})$
    *(*) denotes requested subset of nugget data if applicable*
    $\ell \leftarrow \ell - \| * n_{flakedat} \|$
  $n_{index} \leftarrow n_{index} + 1$
Update and commit metadata and headers
**Ensure:** $\ell = 0$

#### 3.2.2 Merkle Tree

Tracking writes with the transaction journal may stymie a passive attacker by preventing explicit overwrites, but a sufficiently motivated active attacker could resort to all manner of cut-and-paste tactics with nuggets, flakes, and even blocks and sectors. If, for example, an attacker purposefully zeroed-out the transaction journal entry pertaining to a specific nugget in some out-of-band manner—such as when StrongBox is shut down and then later re-initialized with the same backing store—StrongBox would consider any successive incoming writes as if the nugget were in a completely clean state, even though it actually is not. This attack would force StrongBox to make compromising overwrites. To prevent such attacks, we must ensure that the backing store is always in a valid state. More concretely: we must provide an integrity guarantee on top of a confidentiality guarantee.

StrongBox uses our chosen Message Authentication Code (MAC) generating algorithm and each flake's unique key to generate a per-flake MAC tag ("MACed"). The purpose of this tag is to authenticate flake data and confirm that it has not been tampered with. Each tag is then appended to the Merkle Tree along with StrongBox's metadata.

The transaction journal entries are handled specially in that the bit vectors are MACed and the result is appended to the Merkle Tree. This is done to save space.

The Merkle Tree then ties the integrity of any single flake to the integrity of the system as a whole such that if the former fails, *i.e.,* there is a MAC tag mismatch for any particular flake, the latter immediately and obviously fails.

#### 3.2.3 Keycount Store

To prevent a many-time pad attack, each nugget is assigned its own form of nonce we refer to as a *keycount*. The keycount store in Fig. 2 represents a byte-array containing $N$ 8-byte integer keycounts indexed to each nugget. Along with acting as the per-nugget nonce consumed by the stream cipher, the keycount is used to derive the per-flake unique subkeys used in MAC tag generation.

#### 3.2.4 Rekeying Procedure

When a write request would constitute an overwrite, StrongBox will trigger a rekeying process instead of executing the write normally. This rekeying process allows the write to proceed without causing a catastrophic confidentiality violation.

When rekeying begins, the nugget in question is loaded into memory and decrypted. The target data is written into its proper offset in this decrypted nugget. The nugget is then encrypted, this time with a different nonce (*keycount + 1*), and written to the backing store, replacing the outdated nugget data. See: Algorithm 3.

**Algorithm 3** StrongBox rekeying process.

**Require:** The original write applied to a contiguous backing store segment
**Require:** $\ell \leftarrow$ write requested length
**Require:** $\aleph \leftarrow$ master secret
**Require:** $data \leftarrow$ cleartext data to be written
**Require:** $n_{index} \leftarrow$ nugget rekeying target

▷ Read in and decrypt the entire nugget

$n_{nuggetdat} \leftarrow CryptedRead(NSIZE, \aleph, n_{index})$
Calculate indices touched by request: $f_{first}, f_{last}$
Write $data$ into $n_{nuggetdat}$ at proper offset with length $\ell$
Set Transaction Journal entries for $f_{first}, \ldots, f_{last}$ to 1
$k_{n_{index}} \leftarrow GenKey_{nugget}(n_{index}, \aleph)$
Fetch nugget keycount $n_{kc}$ from Keycount Store. Increment it by one.
$n_{nuggetdat} \leftarrow Encrypt(n_{nuggetdat}, k_{n_{index}}, n_{kc})$
Commit $n_{nuggetdat}$ to the backing store

▷ Iterate over all flakes in the nugget

**for all** flakes $f_{current}$ **in** $n_{index}$ **do**
    $k_{f_{current}} \leftarrow GenKey_{flake}(k_{n_{index}}, f_{current}, n_{kc})$
    Copy $f_{current}$ data from $n_{nuggetdat} \rightarrow n_{flakedat}$
    $tag_{f_{current}} \leftarrow GenMac(k_{f_{current}}, n_{flakedat})$
    Update new $tag_{f_{current}}$ in Merkle Tree.
Update and commit metadata and headers

### 3.3 Defending Against Rollback Attacks

To prevent StrongBox from making overwrites, the status of each flake is tracked and overwrites trigger a rekeying procedure. Tracking flake status alone is not enough, however. An attacker could take a snapshot of the backing store in its current state and then easily rollback to a previously valid state. At this point, the attacker could have StrongBox make writes that it does not recognize as overwrites.

With AES-XTS, the threat posed by rolling the backing store to a previously valid state is outside of its threat model. Despite this, data confidentiality guaranteed by AES-XTS holds in the event of a rollback, even if integrity is violated.

StrongBox uses a monotonic global version counter to detect rollbacks. When a rollback is detected, StrongBox will refuse to initialize unless forced, using root permission. Whenever a write request is completed, this global version counter is committed to the backing store, committed to secure hardware, and updated in the in-memory Merkle Tree.

### 3.4 Recovering from Inconsistent State

If StrongBox is interrupted during operation, the backing store—consisting of user data and StrongBox metadata—will be left in an inconsistent state. StrongBox relies on the overlying filesystem *e.g.,* F2FS to manage user-data recovery, which is what these filesystems are designed to do and do well. We detail how StrongBox handles its own inconsistent metadata.

Let $c$ be the value of the on-chip monotonic global version counter and $d$ be the value of the on-drive global version counter header (TPMGLOBALVER). Consider the following:

- *$c == d$ and MTRH is consistent:* StrongBox is operating normally and will mount without issue.

- *$c < d$ or $c == d$ but MTRH is inconsistent:* Since the global version counter is updated before any write, this case cannot be reached unless the backing store was manipulated by an attacker. So, StrongBox will refuse to initialize and cannot be force mounted.
- *$c > d + 1$:* Since the global version counter is updated once per write, this case cannot be reached unless the backing store was rolled back or otherwise manipulated by an attacker. In this case, the root user is warned and StrongBox will refuse to initialize and cannot be force mounted unless the MTRH is consistent. We allow the root user to force mount here if the root user initiated the rollback themselves, such as when recovering from a drive backup.
- *$c == d + 1$:* In this case, StrongBox likely crashed during a write, perhaps during an attempted rekeying. If the rekeying journal is empty or the system cannot complete the rekeying and/or bring the MTRH into a consistent state, the root user is warned and allowed to force mount. Otherwise, StrongBox will not initialize.

For subsequent rekeying efforts in the latter two cases, rather than incrementing the corresponding keystore counters by 1 during rekeying, they will be incremented by 2. This is done to prevent potential reuse of any derived nugget keys that might have been in use right before StrongBox crashed.

When StrongBox can detect tampering, it will not initialize. When StrongBox cannot distinguish between tampering and crash, it offers the root user a choice to force mount. Thus, an attacker could force a crash and use root access to force mount. We assume, however, that if an attacker has root access to a device, its security is already compromised.

# 4 StrongBox Implementation

Our implementation of StrongBox is comprised of 5000 lines of C code. StrongBox uses OpenSSL version 1.0.2 and LibSodium version 1.0.12 for its ChaCha20, Argon2, Blake2, and AES-XTS implementations, likewise implemented in C. The SHA-256 Merkle Tree implementation is borrowed from the Secure Block Device library [18]. StrongBox's implementation is available as open-source.[3]

To reduce the complexity of the experimental setup, establish a fair baseline, and allow StrongBox to run in user space, we use a BUSE [7] virtual block device. BUSE is a thin (200 LOC) wrapper around the standard Linux Network Block Device (NBD), which allows a machine to serve requests for reads and writes to virtual block devices exposed via domain socket. We built StrongBox on top of BUSE/NBD because a simple block device in user space allows for quick experimentation and rapid prototyping. It is not required for a proper implementation.

[3]https://git.xunn.io/research/buselfs-public

## 4.1 Deriving Subkeys

The cryptographic driver requires a shared master secret. The derivation of this master secret is implementation specific and has no impact on performance as it is completed during StrongBox's initialization. Our implementation uses the Argon2 KDF to derive a master secret from a given password with an acceptable time-memory trade-off.

To assign each nugget its own unique keystream, that nugget requires a unique key and associated nonce. We derive these nugget subkeys from the master secret during StrongBox's initialization. To guarantee the backing store's integrity, each flake is tagged with a MAC. We use Poly1305, accepting a 32-byte one-time key and a plaintext of arbitrary length to generate tags. These one-time flake subkeys are derived from their respective nugget subkeys.

## 4.2 A Secure, Persistent, Monotonic Counter

Our target platform uses an embedded Multi-Media Card (eMMC) as a backing store. In addition to boot and user data partitions, the eMMC standard includes a secure storage partition called a Replay Protected Memory Block (RPMB) [10]. The RPMB partition's size is configurable to be at most 16MB (32MB on some Samsung devices) [25]. All read and write commands issued to the RPMB must be authenticated by a key burned into write-once storage (typically eFUSE) during some one-time, secure initialization process.

To implement rollback protection on top of the RPMB, the key for authenticating RPMB commands can be contained in TEE sealed storage or derived from the TPM. For this implementation, StrongBox requires interaction with TPM/TEE secure storage only at mount time, where the authentication key can be retrieved and cached for the duration of StrongBox's lifetime. With the cached key on hand, our implementation makes traditional IOCTL calls to read and write global version counter data to the RPMB eMMC partition, enforcing the invariant that it only increase monotonically.

Our design is not dependent on the eMMC standard, however. Trusted hardware mechanisms other than the eMMC RPMB partition, including TPMs, support secure, persistent storage and/or monotonic counters directly. These can be adapted for use with StrongBox just as well.

There are two practical concerns we must address while implementing the secure counter: wear and performance overhead. Wear is a concern because the counter is implemented in non-volatile storage. The RPMB implements all the same wear protection mechanisms that are used to store user-data [10]. Additionally, StrongBox writes to the global version counter once per write to user-data. Given that the eMMC implements the same wear protection for the RPMB and user data, and that the ratio of writes to these areas is 1:1, we expect StrongBox places no additional wear burden on the hardware. Further, with the JEDEC spec suggesting RPMB implementations use more durable and faster single-level

NAND flash cells rather than cheaper and slower multi-level NAND flash cells [10, 25], the RPMB partition will likely outlive and outperform the user-data portion of the eMMC.

In terms of performance overhead, updating the global version counter requires making one 64-bit authenticated write per user-data write. As user-data writes are almost always substantially larger, we see no significant overhead from the using the RPMB to store the secure counter.

### 4.3 LFS Garbage Collection

An LFS attempts to write to a drive sequentially in an append-only fashion, as if writing to a log. This requires large amounts of contiguous space, called *segments*. Since any backing store is necessarily finite, an LFS can only append so much data before it runs out of space. When this occurs, the LFS triggers a *segment cleaning algorithm* to erase outdated data and compress the remainder of the log into as few segments as possible [22, 27]. This procedure is known more broadly as *garbage collection* [22].

In the context of StrongBox, garbage collection could potentially incur high overhead. The procedure itself would, with its every write, require a rekeying of any affected nuggets. Worse, every proceeding write would appear to StrongBox as if it were an overwrite, since there is no way for StrongBox to know that the LFS triggered garbage collection internally.

In practice, modern production LFSes are optimized to perform garbage collection as few times as possible [22]. Further, they often perform garbage collection in a background thread that triggers when the filesystem is idle and only perform expensive on-demand garbage collection when the backing store is nearing capacity [21, 22]. We leave garbage collection turned on for all of our tests and see no substantial performance degradation from this process because it is scheduled not to interfere with user I/O.

### 4.4 Overhead

StrongBox stores metadata on the drive it is encrypting (see Fig. 3). This metadata should be small compared to the user data. Our implementation uses 4KB flakes, 256 flakes/nugget, and 1024 nuggets per GB of user data. Given the flake and nugget overhead, this configuration requires just over 40KB of metadata per 1 GB of user data. There is an additional, single static header that requires just over 200 bytes. *Thus StrongBox's overhead in terms of storage is less than one hundredth of a percent.*

## 5 Evaluation

### 5.1 Experimental Setup

We implement a prototype of StrongBox on a Hardkernel Odroid XU3 ARM big.LITTLE system (Samsung Exynos5422 A15 and A7 quad core CPUs, 2Gbyte LPDDR3 RAM at 933 MHz, eMMC5.0 HS400 backing store) running Ubuntu Trusty 14.04 LTS, kernel version 3.10.58. The maximum theoretical memory bandwidth for this model is 14.9GB/s. Observed maximum memory bandwidth is 4.5GB/s.

### 5.2 Experimental Methodology

In this section we seek to answer three questions:

1. What is StrongBox's overhead when compared to dm-crypt AES-XTS?
2. How does StrongBox under an LFS (*i.e.,* F2FS) configuration compare to the popular dm-crypt under Ext4 configuration?
3. Where does StrongBox derive its performance gains? Implementation? Choice of cipher?

To evaluate StrongBox's performance, we measure the latency (seconds/milliseconds per operation) of both sequential and random read and write I/O operations across four different standard Linux filesystems: NILFS2, F2FS, Ext4 in ordered journaling mode, and Ext4 in full journaling mode. The I/O operations are performed using file sizes between 4KB and 40MB. These files were populated with random data. The experiments are performed using a 1GB standard Linux ramdisk (tmpfs) as the ultimate backing store.

For sequential F2FS specifically, we include latency measurements dealing with a file size 2.5× the size of available DRAM, *i.e.,* 5GB, supported by a distinct tmpfs backing store swapped into memory.

Ext4's default is ordered journaling mode (`data=ordered`), where metadata is committed to the filesystem's journal while the actual data is written through to the main filesystem. Given a crash, the filesystem uses the journal to avoid damage and recover to a consistent state. Full journaling mode (`data=journal`) journals both metadata and the filesystem's actual data—essentially a double write-back for each write operation. Given a crash, the journal can replay entire I/O events so that both the filesystem and its data can be recovered. We include both modes of Ext4 to further explore the impact of frequent overwrites against StrongBox.

The experiment consists of reading and writing each file in its entirety 30 times sequentially, and then reading and writing random portions of each file 30 times. In both cases, the same amount of data is read and written per file. The median latency is taken per result set. We chose 30 read/write operations (10 read/write operations repeated three times each) to handle potential variation. The Linux page cache is dropped before every read operation, each file is opened in synchronous I/O mode via `O_SYNC`, and we rely on non-buffered `read()`/`write()` system calls. A high-level I/O size of 128KB was used for all read and write calls that hit the filesystems; however, the I/O requests being made at the block device layer varied between 4KB and 128KB depending on the filesystem under test.

The experiment is repeated on each filesystem in three different configurations:

1. *unencrypted*: Filesystem mounted atop a BUSE virtual block device set up to immediately pass through any incoming I/O requests straight to the backing store. We use this as the baseline measurement of the filesystem's performance without any encryption.
2. *StrongBox*: Filesystem mounted atop a BUSE virtual block device provided by our StrongBox implementation to perform full-drive encryption.
3. *dm-crypt*: Filesystem mounted atop a Device Mapper [2] higher-level virtual block device provided by dm-crypt to perform full-drive encryption, which itself is mounted atop a BUSE virtual block device with pass through behavior identical to the device used in the baseline configuration. dm-crypt was configured to use AES-XTS as its full-drive encryption algorithm. All other parameters were left at their default values.

Fig. 4 compares StrongBox to dm-crypt under the F2FS filesystem. The gamut of result sets over different filesystems can be seen in Fig. 5. Fig. 6 compares Ext4 with dm-crypt to F2FS with StrongBox.

### 5.3 StrongBox Read Performance

Fig. 4 shows the performance of StrongBox in comparison to dm-crypt, both mounted with the F2FS filesystem. We see StrongBox improves on the performance of dm-crypt's AES-XTS implementation across sequential and random read operations on all file sizes. Specifically, 2.36× (53.05m/22.48m) for sequential 5GB, 2.07× (2.09s/1.00s) for sequential 40MB, 2.08× (267.34ms/128.22ms) for sequential 5MB, 1.85× (28.30ms/15.33ms) for sequential 512KB, and 1.03× (0.95ms/0.86ms) for sequential 4KB.

Fig. 5 provides an expanded performance profile for StrongBox, testing a gamut of filesystems broken down by workload file size. For sequential reads across all filesystems and file sizes, StrongBox outperforms dm-crypt. This is true even on the non-LFS Ext4 filesystems. Specifically, we see read performance improvements over dm-crypt AES-XTS for 40MB sequential reads of 2.02× (2.15s/1.06s) for NILFS, 2.07× (2.09s/1.00s) for F2FS, 2.09× (2.11s/1.01s) for Ext4 in ordered journaling mode, and 2.06× (2.11s/1.02s) for Ext4 in full journaling mode. For smaller file sizes, the performance improvement is less pronounced. For 4KB reads we see 1.28× (1.62ms/1.26ms) for NILFS, 1.03× (0.88ms/0.86ms) for F2FS, 1.04× (0.95ms/0.92ms) for Ext4 in ordered journaling mode, and 1.07× (0.97ms/0.91ms) for Ext4 in full journaling mode. When it comes to random reads, we see virtually identical results save for 4KB reads, where dm-crypt proved slightly more performant under the NILFS LFS at 1.12× (1.73ms/1.54ms). This behavior is not observed with the more modern F2FS.

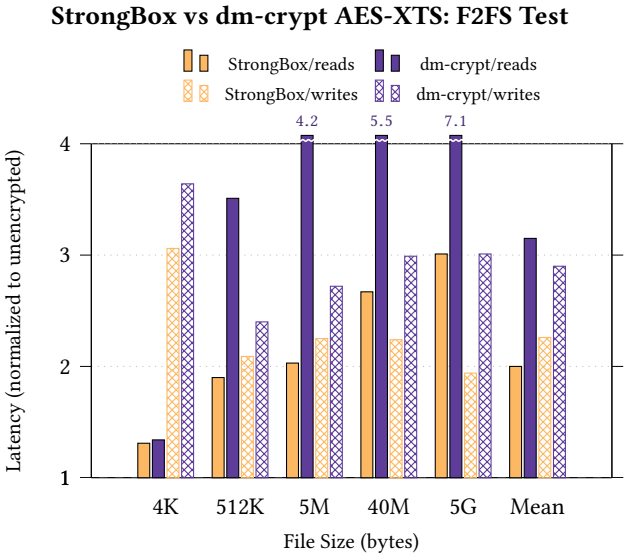


Figure **4.a:** Sequential I/O expanded F2FS result set.

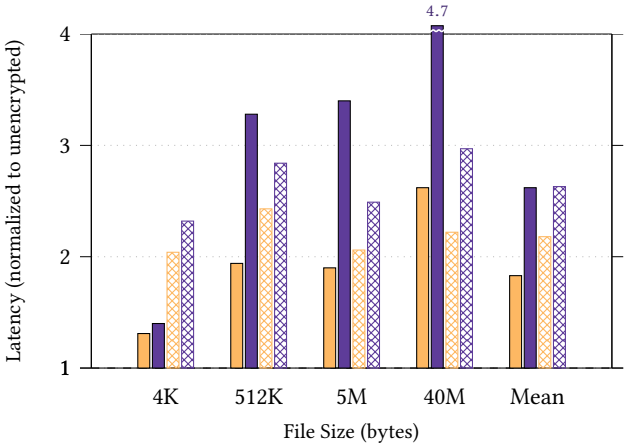


Figure **4.b:** Random I/O expanded F2FS result set.

**Figure 4.** Test of the F2FS LFS mounted atop both dm-crypt and StrongBox; median latency of different sized whole file read and write operations normalized to unencrypted access. By harmonic mean, StrongBox is 1.72× faster than dm-crypt for sequential reads and 1.27× faster for sequential writes.

### 5.4 StrongBox Write Performance

Fig. 4 shows the performance of StrongBox in comparison to dm-crypt under the modern F2FS LFS broken down by workload file size. Similar to read performance under the F2FS, we see StrongBox improves on the performance of dm-crypt's AES-XTS implementation across sequential and random write operations on all file sizes. Hence, StrongBox under F2FS is holistically faster than dm-crypt under F2FS. Specifically, 1.55× (1.80h/1.16h) for sequential 5GB, 1.33× (3.19s/2.39s) for sequential 40MB, 1.21× (412.51ms/341.56ms)

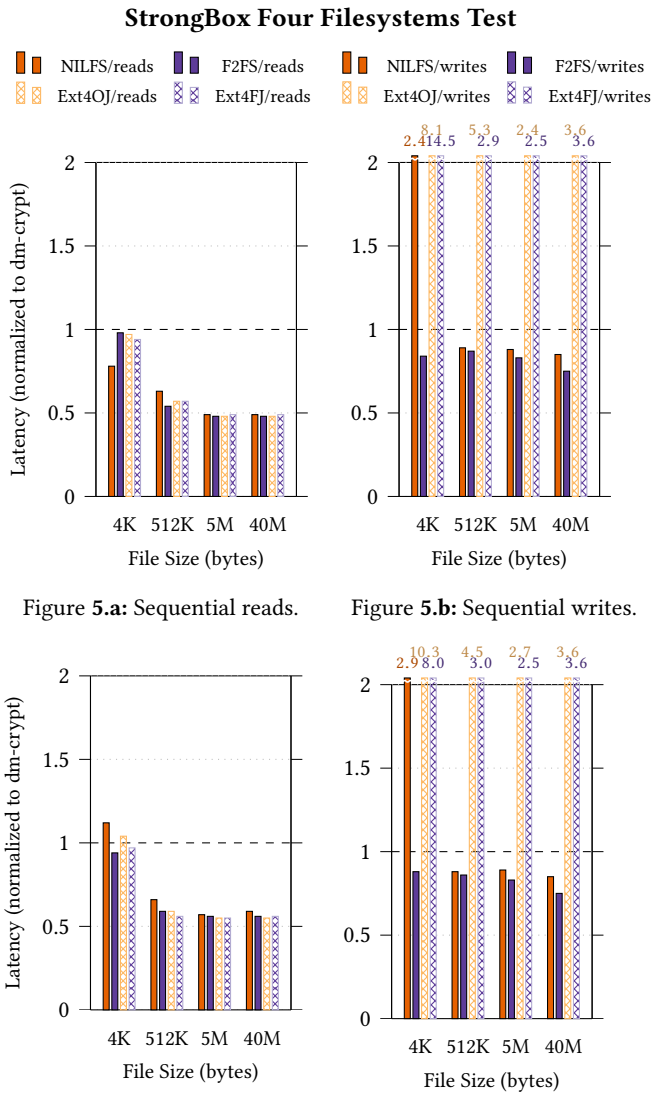


Figure **5.a:** Sequential reads.

Figure **5.b:** Sequential writes.

Figure **5.c:** Random reads.

Figure **5.d:** Random writes.

**Figure 5.** Comparison of four filesystems running on top of StrongBox performance is normalized to the same file system running on dm-crypt. Points below the line signify StrongBox outperforming dm-crypt. Points above the line signify dm-crypt outperforming StrongBox.

for sequential 5MB, 1.15× (65.23ms/56.63ms) for sequential 512KB, and 1.19× (30.30ms/25.46ms) for sequential 4KB.

Fig. 5 provides an expanded performance profile for StrongBox, testing a gamut of filesystems broken down by workload file size. Unlike read performance, write performance under certain filesystems is more of a mixed bag. For 40MB sequential writes, StrongBox outperforms dm-crypt's AES-XTS implementation by 1.33× (3.19s/2.39s) for F2FS and 1.18× (4.39s/3.74s) for NILFS. When it comes to Ext4, StrongBox's write performance drops precipitously with a 3.6× *slowdown* for both ordered journaling and full journaling modes (respectively: 12.64s/3.51s, 24.89s/6.88s). For non-LFS 4KB writes, the performance degradation is even more pronounced with a 8.09× (118.48ms/14.65ms) slowdown for ordered journaling and 14.5× (143.15ms/9.87ms) slowdown for full journaling.

This slowdown occurs in Ext4 because, while writes in StrongBox from non-LFS filesystems have a metadata overhead that is comparable to that of forward writes in an LFS filesystem, Ext4 is not an append-only or append-mostly filesystem. This means that, at any time, Ext4 will initiate one or more overwrites anywhere on the drive (see Table 1). As described in Section 3, overwrites once detected trigger the rekeying process, which is a relatively expensive operation. Multiple overwrites compound this expense further. This makes Ext4 and other filesystems that do not exhibit at least append-mostly behavior unsuitable for use with StrongBox. We include it in our result set regardless to illustrate the drastic performance impact of frequent overwrites on StrongBox.

For both sequential and random 4KB writes among the LFSes, the performance improvement over dm-crypt's AES-XTS implementation for LFSes deflates. For the more modern F2FS atop StrongBox, there is a 1.19× (30.30ms/25.46ms) improvement. For the older NILFS filesystem atop StrongBox, there is a 2.38× (27.19ms/11.44ms) slowdown. This is where we begin to see the overhead associated with tracking writes and detecting overwrites potentially becoming problematic, though the overhead is negligible depending on choice of LFS and workload characteristics.

These results show that StrongBox is sensitive to the behavior of the LFS that is mounted atop it, and that any practical use of StrongBox would require an extra profiling step to determine which LFS works best with a specific workload. With the correct selection of LFS, such as F2FS for workloads dominated by small write operations, potential slowdowns when compared to mounting that same filesystem over dm-crypt's AES-XTS can be effectively mitigated.

### 5.5 On Replacing dm-crypt and Ext4

Fig. 6 describes the performance benefit of using StrongBox with F2FS over the popular dm-crypt with Ext4 in ordered journaling mode combination for both sequential and random read and write operations of various sizes. Other than 4KB and 512KB write operations, which are instances where baseline F2FS without StrongBox is simply slower than baseline Ext4 without dm-crypt, StrongBox with F2FS outperforms dm-crypt's AES-XTS implementation with Ext4.

These results show that configurations taking advantage of the popular combination of dm-crypt, AES-XTS, and Ext4 could see a significant improvement in read performance without a degradation in write performance except in cases where small ($\leq 512KB$) writes dominate the workload.

Note, however, that several implicit assumptions exist in our design. For one, we presume there is ample memory at

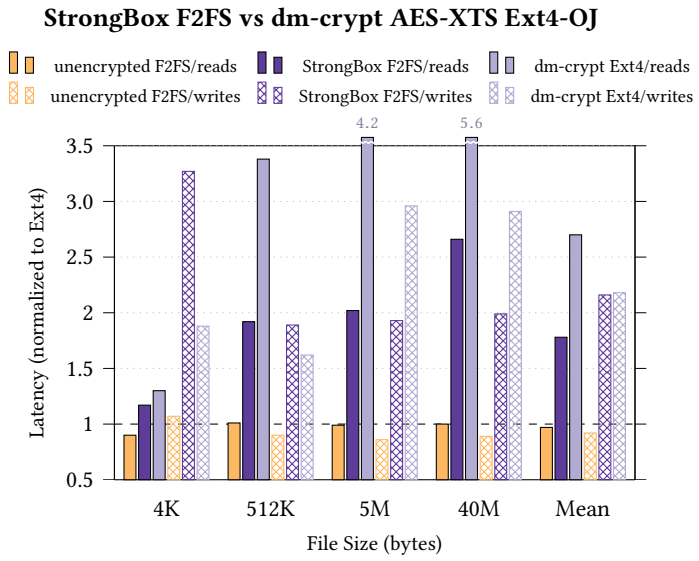


Figure **6.a:** Sequential I/O F2FS vs Ext4 result set.

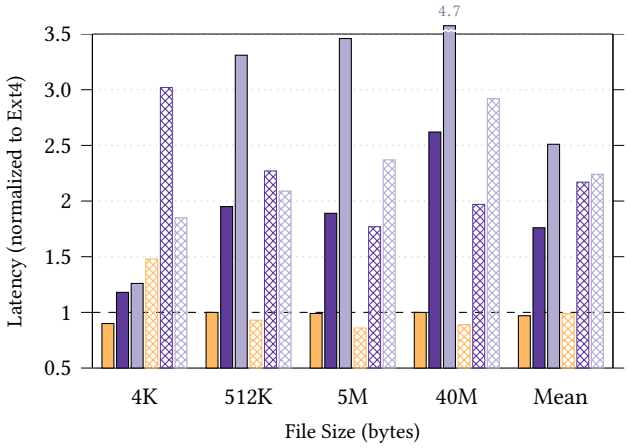


Figure **6.b:** Random I/O F2FS vs Ext4 result set.

**Figure 6.** Comparison of Ext4 on dm-crypt and F2FS on StrongBox. Results are normalized to unencrypted Ext4 performance. Unecrypted F2FS results are shown for reference. Points below the line are outperforming unencrypted Ext4. Points above the line are underperforming compared to unencrypted Ext4.

hand to house the Merkle Tree and all other data abstractions used by StrongBox. Efficient memory use was not a goal of our implementation of StrongBox. In an implementation aiming to be production ready, much more memory efficient data structures would be utilized.

It is also for this reason that populating the Merkle Tree necessitates a rather lengthy mounting process. In our tests, a 1GB backing store on the odroid system can take as long as 15 seconds to mount.

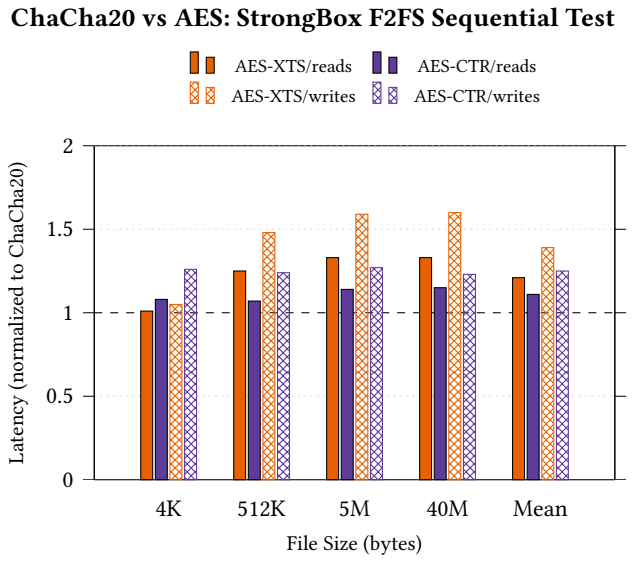


**Figure 7.** Comparison of AES in XTS and CTR modes versus ChaCha20 in StrongBox; median latency of different sized whole file sequential read and write operations normalized to ChaCha20 (default cipher in StrongBox). Points below the line signify AES outperforming ChaCha20. Points above the line signify ChaCha20 outperforming AES.

### 5.6 Performance in StrongBox: ChaCha20 vs AES

Fig. 5 and Fig. 4 give strong evidence for our general performance improvement over dm-crypt not being an artifact of filesystem choice. Excluding Ext4 as a non-LFS filesystem under which to run StrongBox, our tests show that StrongBox outperforms dm-crypt under an LFS filesystem in the vast majority of outcomes.

We then test to see if our general performance improvement can be attributed to the use of a stream cipher over a block cipher. dm-crypt implements AES in XTS mode to provide full-drive encryption functionality. Fig. 7 describes the relationship between ChaCha20, the cipher of choice for our implementation of StrongBox, and the AES cipher. Swapping out ChaCha20 for AES-CTR resulted in slowdowns of up to 1.33× for reads and 1.15× for writes across all configurations, as described in Fig. 7.

Finally, we test to see if our general performance improvement can be attributed to our implementation of StrongBox rather than our choice of stream cipher. We test this by implementing AES in XTS mode on top of StrongBox using OpenSSL EVP (see: Fig. 7). StrongBox using OpenSSL AES-XTS experiences slowdowns of up to 1.33× for reads and 1.6× for writes compared to StrongBox using ChaCha20 (sequential, 40MB). Interestingly, while significantly less performant, this slowdown is not entirely egregious, and suggests that perhaps there are parts of the dm-crypt code base that would benefit from further optimization; however, it is possible

that necessary choices to harden StrongBox for a production environment could slow it down as well.

Considering hardware support for dedicated AES instructions, Fig. 7 shows StrongBox with AES-CTR outperforms AES-XTS. Therefore, StrongBox should still outperform dm-crypt where AES hardware support is available.

### 5.7 Overhead with a Full Drive

During I/O operations under an appropriate choice of LFS, we have shown that full-drive encryption provided by StrongBox outperforms full-drive encryption provided by dm-crypt. However, this is not necessarily the case when the backing store becomes full and the LFS is forced to cope with an inability to write forward as efficiently.

In the case of the F2FS LFS, upon approaching capacity and being unable to perform garbage collection effectively, it resorts to writing blocks out to where ever it can find free space in the backing store [22]. It does this instead of trying to maintain an append-only guarantee. This method of executing writes is similar to how a typical non-LFS filesystem operates. When this happens, the F2FS aggressively causes overwrites in StrongBox, which has a drastic impact on performance.

Fig. 8 shows the impact of these (sequential) overwrites. Read operation performance remains faster on a full StrongBox backing store compared to dm-crypt. This is not the case with writes. Compared to StrongBox under non-full conditions, 40MB sequential writes were slowed by 2.8× as StrongBox approached maximum capacity.

### 5.8 Threat Analysis

Table 2 lists possible attacks and their results. It can be inferred from these results and StrongBox's design that StrongBox addresses its threat model and maintains confidentiality and integrity guarantees.

## 6 Related Work

Some of the most popular cryptosystems offering a confidentiality guarantee for data at rest employ a symmetric encryption scheme known as a Tweakable Enciphering Scheme (TES) [13, 26]. There have been numerous TES-based constructions securing data at rest [13, 17, 33], including the well known XEX-based XTS operating mode of AES [4] explored earlier in this work. Almost all TES constructions and the storage management systems that implement them use one or more block ciphers as their primary primitive [13, 28].

Our StrongBox implementation borrows from the design of these systems. One in particular is *dm-crypt*, a Linux framework employing a *LinuxDeviceMapper* to provide a virtual block interface for physical block devices. Dm-crypt provides an implementation of the AES-XTS algorithm among others and is used widely in the Linux ecosystem [1, 8]. The algorithms provided by dm-crypt all employ block ciphers [8].

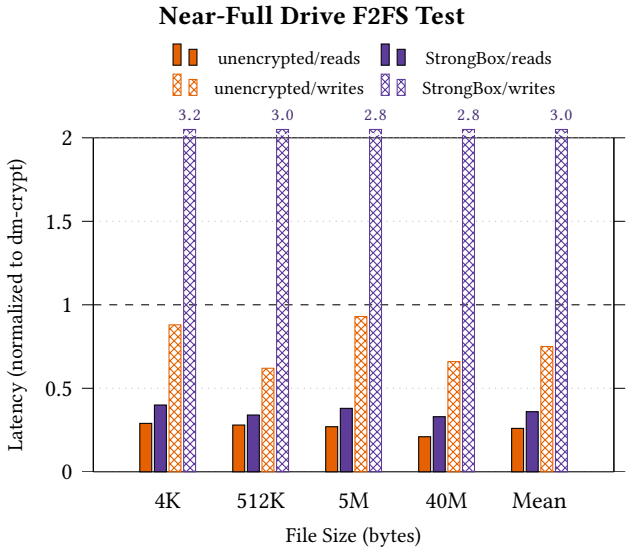


**Figure 8.** Comparison of F2FS baseline, atop dm-crypt, and atop StrongBox. All configurations are initialized with a near-full backing store; median latency of different sized whole file read and write operations normalized to dm-crypt. Points below the line are outperforming dm-crypt. Points above the line are underperforming compared to dm-crypt.

**Table 2.** Attacks on StrongBox and their results

| Attack | Result | Explanation |
|---|---|---|
| Nugget user data in backing store is mutated out-of-band online | StrongBox Immediately fails with exception on successive IO request | The MTRH is inconsistent |
| Header metadata in backing store is mutated out-of-band online, making the MTRH inconsistent | StrongBox Immediately fails with exception on successive IO request | The MTRH is inconsistent |
| Backing store is rolled back to a previously consistent state while online | StrongBox Immediately fails with exception on successive IO request | TPMGLOBALVER and RPMB secure counter out of sync |
| Backing store is rolled back to a previously consistent state while offline, RPMB secure counter wildly out of sync | StrongBox refuses to mount; allows for force mount with root access | TPMGLOBALVER and RPMB secure counter out of sync |
| MTRH made inconsistent by mutating backing store out-of-band while offline, RPMB secure counter in sync | StrongBox refuses to mount | TPMGLOBALVER and RPMB secure counter are in sync, yet illegal data manipulation occurred |

Instead of a block cipher, however, StrongBox uses a stream

cipher to provide the same confidentiality guarantee and consistent or better I/O performance. Further unlike dm-crypt and other similar virtualization frameworks, StrongBox's ciphering operations do not require sector level tweaks, depending on the implementation. With StrongBox, several physical blocks consisting of one or more sectors are considered as discrete logical units, *i.e.,* nuggets and flakes.

Substituting a block cipher for a stream cipher forms the core of several contributions to the state-of-the-art [13, 28]. Chakraborty et al. proposed STES—a stream cipher based low cost scheme for securing stored data [13]. STES is a novel TES which can be implemented compactly with low overall power consumption. It combines a stream cipher and a universal hash function via XOR and is targeting low cost FPGAs to provide confidentiality of data on USBs and SD cards. Our StrongBox, on the other hand, is not a TES and does not directly implement a TES. StrongBox combines a stream cipher with nonce "tweak" and nugget data via XOR and is targeting any configuration employing a well-behaved Log-structured Filesystem (LFS) at some level to provide confidentiality of data.

Offering a transparent cryptographic layer at the block device level has been proposed numerous times [18]. Production implementations include storage management systems like dm-crypt. Specifically, Hein et al. proposed the Secure Block Device (SBD) [18]—an ARM TrustZone secure world transparent filesystem encryption layer optimized for ANDIX OS and implemented and evaluated using the Linux Network Block Device (NBD) driver. StrongBox is also implemented and evaluated using the NBD, but is not limited to one specific operating system. Further unlike StrongBox, SBD is not explicitly designed for use outside of the ARM TrustZone secure world. Contrarily, StrongBox was designed to be used on any system that provides a subset of functionality provided by a Trusted Platform Module (TPM) and/or Trusted Execution Environment (TEE). Specifically, StrongBox requires the availability of a dedicated hardware protected secure monotonic counter to prevent rollback attacks and ensure the freshness of StrongBox. The primary design goal of StrongBox is to achieve provide higher performance than the industry standard AES-XTS algorithm utilizing a stream cipher.

StrongBox's design is only possible because of the availability of hardware support for security, which has been a major thrust of research efforts [15, 19, 23, 30, 31, 34], and is now available in almost all commercial mobile processors [16, 20, 25, 29]. Our implementation makes use of the replay protected memory block on eMMC devices [10, 25], but it could be reimplemented using any hardware that supports persistent, monotonic counters.

The combination of trusted hardware and monotonic counters enables new security mechanisms. For example, van Dijk et al. use this combination allow clients to securly store data on an untrusted server [32]. Like StrongBox, their approach relies on trusted hardware (TPM specifically [16]), logs, and monotonic counters. The van Dijk et al. approach, however, uses existing secure storage and is not concerned with storage speed. StrongBox uses these same mechanisms along with novel metadata layout and system design to solve a different problem: providing higher performance than AES-XTS based approaches.

Achieving on-drive data integrity protection through the use of checksums has been used by filesystems and many other storage management systems. Examples include ZFS [3] and others [18]. For our implementation of StrongBox, we used the Merkle Tree library offered by SBD to manage our in-memory checksum verification. A proper implementation of StrongBox need not use the SDB SHA-256 Merkle Tree library. It was chosen for convenience.

## 7 Conclusion

The conventional wisdom is that securing data at rest requires one must pay the high performance overhead of encryption with AES is XTS mode. This paper shows that technological trends overturn this conventional wisdom: Log-structured file systems and hardware support for secure counters make it practical to use a stream cipher to secure data at rest. We demonstrate this practicality through our implementation of StrongBox which uses the ChaCha20 stream cipher and the Poly1305 MAC to provide secure storage and can be used as a drop-in replacement for dm-crypt.

Our empirical results show that under F2FS—a modern, industrial-strength Log-structured file system—StrongBox provides upwards of 2× improvement on read performance and average 1.27× improvement on write performance compared to dm-crypt. Further, our results show that F2FS plus StrongBox provides a higher performance replacement for Ext4 backed with dm-crypt. We make our implementation of StrongBox available open source so that others can extend it or compare to it.[4] Our hope is that this work motivates further exploration of fast stream ciphers as replacements for AES-XTS for securing data at rest.

## Acknowledgments

We would like to thank the anonymous reviewers for their insightful feedback and comments. This material is based upon work supported by the National Science Foundation under Grant No. CNS-1526304. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation.

[4] https://git.xunn.io/research/buselfs-public

## References


[1] [n. d.]. Android Open Source Project: Full-Disk Encryption. ([n. d.]). https://source.android.com/security/encryption/full-disk

[2] [n. d.]. RedHat: Device-mapper Resource Page. ([n. d.]). https://www.sourceware.org/dm

[3] 2005. Oracle blog: ZFS End-to-End Data Integrity. (2005). https://blogs.oracle.com/bonwick/zfs-end-to-end-data-integrity

[4] 2008. The XTS-AES Tweakable Block Cipher. (2008). IEEE Std 1619-2007.

[5] 2010. Recommendation for Block Cipher Modes of Operation: The XTS-AES Mode for Confidentiality on Storage Devices. (2010). http://nvlpubs.nist.gov/ NIST Special Publication 800-38E.

[6] 2011. Message Authentication Code Standard ISO/IEC 9797-1:2011. (2011). https://www.iso.org/standard/50375.html

[7] 2012. A block device in userspace. (2012). https://github.com/acozzette/BUSE

[8] 2013. Linux kernel device-mapper crypto target. (2013). https://gitlab.com/cryptsetup/cryptsetup

[9] 2014. TLS Symmetric Crypto. (2014). https://www.imperialviolet.org/2014/02/27/tlssymmetriccrypto.html

[10] 2015. EMBEDDED MULTI-MEDIA CARD (e•MMC), ELECTRICAL STANDARD (5.1). (2015). https://www.jedec.org/standards-documents/results/jesd84-b51

[11] Daniel J. Bernstein. 2005. *The Poly1305-AES message-authentication code.* Technical Report. University of Illinois at Chicago.

[12] Daniel J. Bernstein. 2008. *ChaCha, a variant of Salsa20.* Technical Report. University of Illinois at Chicago.

[13] D. Chakraborty, C. Mancillas-López, and P. Sarkar. 2015. STES: A Stream Cipher Based Low Cost Scheme for Securing Stored Data. *IEEE Trans. Comput.* 64, 9 (2015), 2691–2707. https://doi.org/10.1109/TC.2014.2366739

[14] Michael Cornwell. 2012. Anatomy of a Solid-state Drive. *Queue* 10, 10, Article 30 (Oct. 2012), 7 pages. https://doi.org/10.1145/2381996.2385276

[15] Andrew Ferraiuolo, Rui Xu, Danfeng Zhang, Andrew C. Myers, and G. Edward Suh. 2017. Verification of a Practical Hardware Security Architecture Through Static Information Flow Analysis. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2017, Xi'an, China, April 8-12, 2017.* 555–568. https://doi.org/10.1145/3037697.3037739

[16] Trusted Computing Group. 2008. TCG: Trusted platform module summary. (2008).

[17] Shai Halevi and Phillip Rogaway. 2003. *A Tweakable Enciphering Mode.* Springer Berlin Heidelberg, Berlin, Heidelberg, 482–499. https://doi.org/10.1007/978-3-540-45146-4_28

[18] D. Hein, J. Winter, and A. Fitzek. 2015. Secure Block Device – Secure, Flexible, and Efficient Data Storage for ARM TrustZone Systems. In *2015 IEEE Trustcom/BigDataSE/ISPA*, Vol. 1. 222–229. https://doi.org/10.1109/Trustcom.2015.378

[19] Matthew Hicks, Cynthia Sturton, Samuel T. King, and Jonathan M. Smith. 2015. SPECS: A Lightweight Runtime Mechanism for Protecting Software from Security-Critical Processor Bugs. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '15, Istanbul, Turkey, March 14-18, 2015.* 517–529. https://doi.org/10.1145/2694344.2694366

[20] Darko Kirovski, Milenko Drinić, and Miodrag Potkonjak. 2002. Enabling Trusted Software Integrity. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS X).* ACM, New York, NY, USA, 108–120. https://doi.org/10.1145/605397.605409

[21] Ryusuke Konishi, Yoshiji Amagai, Koji Sato, Hisashi Hifumi, Seiji Kihara, and Satoshi Moriai. 2006. The Linux Implementation of a Log-structured File System. *SIGOPS Oper. Syst. Rev.* 40, 3 (July 2006), 102–107. https://doi.org/10.1145/1151374.1151375

[22] Changman Lee, Dongho Sim, Jooyoung Hwang, and Sangyeun Cho. 2015. F2FS: A New File System for Flash Storage. In *13th USENIX Conference on File and Storage Technologies (FAST 15).* USENIX Association, Santa Clara, CA, 273–286. https://www.usenix.org/conference/fast15/technical-sessions/presentation/lee

[23] Xun Li, Vineeth Kashyap, Jason K. Oberg, Mohit Tiwari, Vasanth Ram Rajarathinam, Ryan Kastner, Timothy Sherwood, Ben Hardekopf, and Frederic T. Chong. 2014. Sapper: a language for hardware-level security policy enforcement. In *Architectural Support for Programming Languages and Operating Systems, ASPLOS '14, Salt Lake City, UT, USA, March 1-5, 2014.* 97–112. https://doi.org/10.1145/2541940.2541947

[24] ARM Limited. 2009. ARM security technology: Building a secure system using TrustZone technology. (2009). PRD29-GENC-009492C.

[25] Anil Kumar Reddy, P. Paramasivam, and Prakash Babu Vemula. 2015. Mobile secure data protection using eMMC RPMB partition. In *2015 International Conference on Computing and Network Communications (CoCoNet).* 946–950. https://doi.org/10.1109/CoCoNet.2015.7411305

[26] Phillip Rogaway. 2004. *Efficient Instantiations of Tweakable Blockciphers and Refinements to Modes OCB and PMAC.* Technical Report. University of California at Davis.

[27] Mendel Rosenblum and John K. Ousterhout. 1992. The Design and Implementation of a Log-structured File System. *ACM Trans. Comput. Syst.* 10, 1 (Feb. 1992), 26–52. https://doi.org/10.1145/146941.146943

[28] Palash Sarkar. 2009. *Tweakable Enciphering Schemes From Stream Ciphers With IV.* Technical Report. Indian Statistical Institute.

[29] Global Platform Device Technology. 2010. TEE client API specification version 1.0. (2010). GPD_SPE_007.

[30] Mohit Tiwari, Jason Oberg, Xun Li, Jonathan Valamehr, Timothy E. Levin, Ben Hardekopf, Ryan Kastner, Frederic T. Chong, and Timothy Sherwood. 2011. Crafting a usable microkernel, processor, and I/O system with strict and provable information flow security. In *38th International Symposium on Computer Architecture (ISCA 2011), June 4-8, 2011, San Jose, CA, USA.* 189–200. https://doi.org/10.1145/2000064.2000087

[31] Jonathan Valamehr, Melissa Chase, Seny Kamara, Andrew Putnam, Daniel Shumow, Vinod Vaikuntanathan, and Timothy Sherwood. 2012. Inspection resistant memory: Architectural support for security from physical examination. In *39th International Symposium on Computer Architecture (ISCA 2012), June 9-13, 2012, Portland, OR, USA.* 130–141. https://doi.org/10.1109/ISCA.2012.6237012

[32] Marten van Dijk, Jonathan Rhodes, Luis F. G. Sarmenta, and Srinivas Devadas. 2007. Offline Untrusted Storage with Immediate Detection of Forking and Replay Attacks. In *Proceedings of the 2007 ACM Workshop on Scalable Trusted Computing (STC '07).* ACM, New York, NY, USA, 41–48. https://doi.org/10.1145/1314354.1314364

[33] Peng Wang, Dengguo Feng, and Wenling Wu. 2005. *HCTR: A Variable-Input-Length Enciphering Mode.* Springer Berlin Heidelberg, Berlin, Heidelberg, 175–188. https://doi.org/10.1007/11599548_15

[34] Rui Zhang, Natalie Stanley, Christopher Griggs, Andrew Chi, and Cynthia Sturton. 2017. Identifying Security Critical Properties for the Dynamic Verification of a Processor. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2017, Xi'an, China, April 8-12, 2017.* 541–554. https://doi.org/10.1145/3037697.3037734