THE UNIVERSITY OF CHICAGO

STRONGBOX: CONFIDENTIALITY, INTEGRITY, AND PERFORMANCE USING STREAM CIPHERS FOR FULL-DISK ENCRYPTION

A DISSERTATION SUBMITTED TO
THE FACULTY OF THE DIVISION OF THE PHYSICAL SCIENCES
IN CANDIDACY FOR THE DEGREE OF
MASTER

DEPARTMENT OF COMPUTER SCIENCE DEPARTMENT

BY
BERNARD DICKENS III

CHICAGO, ILLINOIS

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under Grant No. CNS-1526304. I would also like to acknowledge my advisors, Henry Hoffmann and Ariel Feldman, without whom this work would not have been possible.

ABSTRACT

Full disk encryption (FDE) is especially important for mobile devices because they both contain large amounts of sensitive data and are easily lost or stolen. Yet, the conventional approach to FDE, AES in XTS mode, is 3–5× slower than unencrypted storage. Authenticated encryption based on stream ciphers like ChaCha20 is already used as a faster alternative to AES in other contexts, such as HTTPS, but the conventional wisdom is that stream ciphers are a unsuitable for FDE. Used naively in disk encryption, stream ciphers are vulnerable to many-time pad attacks and rollback attacks, and mitigating these attacks with on-disk metadata is generally believed to ruin performance.

In this paper, we argue that recent developments in mobile devices invalidate this assumption and make it possible to use fast stream ciphers for disk encryption. Modern mobile devices use Log-structured File Systems and 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 disk encryption module based on AES-XTS. StrongBox introduces a system design and on-disk data structures that exploit LFS's lack of overwrites to avoid costly rekeying and a counter stored in trusted hardware to implement rollback protection. We implement StrongBox on an ARM big.LITTLE mobile processor and test its performance under multiple popular production LFSes. We find that StrongBox generally improves read performance by over 1.6× and write performance by over 1.2× compared to dm-crypt while offering stronger integrity guarantees.

CHAPTER 1

INTRODUCTION

Full disk encryption (FDE) 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 on such devices is to use the AES cipher in XTS mode [16, 22]. 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 [4], can be faster than using AES (see Section 2). Indeed in 2014, Google made the case for stream ciphers over AES, switching HTTPS connections on Chrome for Android to use a stream cipher for better performance [23]. Stream ciphers are not used for full disk encryption, however, because when applied naively, they are trivially vulnerable to *many-time pad attacks* and *rollback attacks* that can reveal the plaintext. Furthermore, it has been assumed that full disk encryption should be transparent and length-preserving (*i.e.,* that every sector should be encrypted independently and no extra space should be reserved for MAC tags) [22] because doing otherwise would ruin performance. Thus, the conventional wisdom is that full disk encryption necessarily incurs the overhead of AES-XTS or a similar primitive.

In this paper, we argue that two technological shifts in mobile devices overturn this conventional wisdom and make it possible to use fast stream ciphers for disk encryption. First, mobile devices commonly use Flash Translation Layer (FTL) managed NAND-flash (i.e. eMMC) and/or Log-structured File Systems (LFSes) [10, 11, 19] to increase the lifetime of their solid-state drives (SSDs). Second, modern mobile devices like smartphones now come equipped with trusted hardware, such as Trusted Execution Environments (TEEs) [12, 21], and secure storage areas [7]. The use of LFSes means that overwrites to the same disk sectors will be limited, with most writes simply appended to a log. Having fewer overwrites dramatically improves the performance of using stream ciphers, which require an expensive

re-key on every overwrite. The presence of secure hardware means that disk encryption modules have access to persistent, monotonically increasing counters that can be used to prevent rollback attacks.

Given these trends, we propose StrongBox, a new method for securing data at rest designed as a drop-in replacement for AES-XTS-backed FDE providers such as dm-crypt [13]; *i.e.,* it requires no interface changes. StrongBox combines authenticated encryption based on a fast stream cipher with a secure, persistent counter—supported by existing mobile hardware—to prevent rollback attacks. The challenge of building StrongBox is that even with Log-structured File Systems running on or above SSDs, blocks will occasionally be overwritten; *e.g.,* by segment cleaning or *garbage collection* in the LFS. *Thus, StrongBox's main contribution is a system design and a set of on-disk data structures that enable secure disk encryption with a stream cipher without compromising performance.*

We demonstrate StrongBox's effectiveness by implementing it 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 [4] as our chosen stream cipher and Poly1305 [3] as our chosen MAC algorithm. As StrongBox requires no change to any existing interfaces, we benchmark it on two of the most popular LFSes: NILFS [10] and F2FS [11]. We compare the performance of these systems on top of AES-XTS (via dm-crypt) and StrongBox. Additionally, we compare to the performance of AES-XTS encrypted Ext4 filesystems with StrongBox. 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; *i.e.,* under F2FS, StrongBox provides a 1.6× speedup over AES-XTS.

- *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; *i.e.,* under F2FS,

StrongBox provides a 1.2× 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 [22], StrongBox provides the chosen message attack security of standard authenticated encryption. The remaining sections of this paper motivate the use of stream ciphers in greater detail, present StrongBox's design, argue for its security, and empirically evaluate its performance in comparison with the state-of-the-art. In addition, StrongBox's implementation will be made available.[1]

1. https://github.com/ananonrepo2/StrongBox

CHAPTER 2

MOTIVATION

The section reviews the two main motivations for StrongBox: the relative speed of stream ciphers compared to AES-XTS and the append-mostly nature of Log-structured File Systems. 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. 2.1 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 allow an attacker to trivially recover the secret key. Thus we believe stream ciphers are best suited for encrypting block devices backing Log-structured File Systems, 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.

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

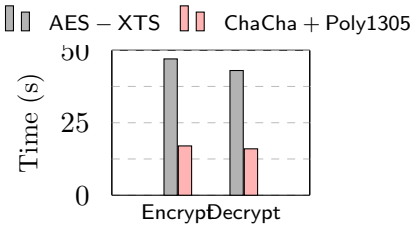


Figure 2.1: AES-XTS and ChaCha20+Poly1305 Comparison.

total writes to the underlying block device and the number of times data is overwritten for each file system.

Table 2.1: 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 2.1 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

The above data motivates our approach of using a stream cipher to perform full disk encryption underneath Log-structured File Systems. The stream cipher is more than twice as fast as AES-XTS. The problem with the stream cipher is that it is not secure if the same key is used to overwrite the same storage location. Fortunately, under normal operation, the log-structured file systems rarely overwrite the same location.

We cannot, however, ignore the fact that overwrites do occur. The data in this section shows that overwrites are rare during normal operation. We also know they will be common when garbage collecting the Log-structured File System. Thus, we will need some metadata

to help us track writes and ensure that data is re-keyed if overwrites occur. Therefore, we recognize three challenges to replacing AES with ChaCha20 for full disk encryption:

- 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 side channel leaks or rollback attacks.
- Accomplishing the above efficiently so that we maintain the performance advantage of the stream cipher.

The key to our approach is using a secure, persistent counter supported in modern mobile hardware; *e.g.,* for limited password entry attempts. The intuition is that this counter can be used to 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 system would detect that the local version number is out of sync with the global version number stored in the secure counter, StrongBox would refuse to initialize, and the attack would fail. 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.

CHAPTER 3

STRONGBOX SYSTEM DESIGN

StrongBox acts as a translation layer sitting between the disk 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) [6].

Hence, there are several locations where StrongBox could be implemented in the system stack. StrongBox could be integrated into an LFS filesystem 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, which is where we chose to implement our prototype. StrongBox could even be implemented within the on-disk SSD controller responsible for managing the flash translation layer (scatter gather, garbage collection, wear-leveling, etc.) on modern SSDs and other types of non-volatile storage.

StrongBox's design is illustrated in Fig. 3.1. StrongBox's metadata is encapsulated in three primary components: an in-memory *Merkle Tree* and two disk-backed byte arrays, the *Keycount Store* and the *Transaction Journal*. These components are integrated into the *Cryptographic Driver*, which is responsible for handling 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 overlying LFS. Low-level I/O between StrongBox and the backing store is handled by the *Device Controller*.

The rest of this section describes the components referenced in Fig. 3.1. 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

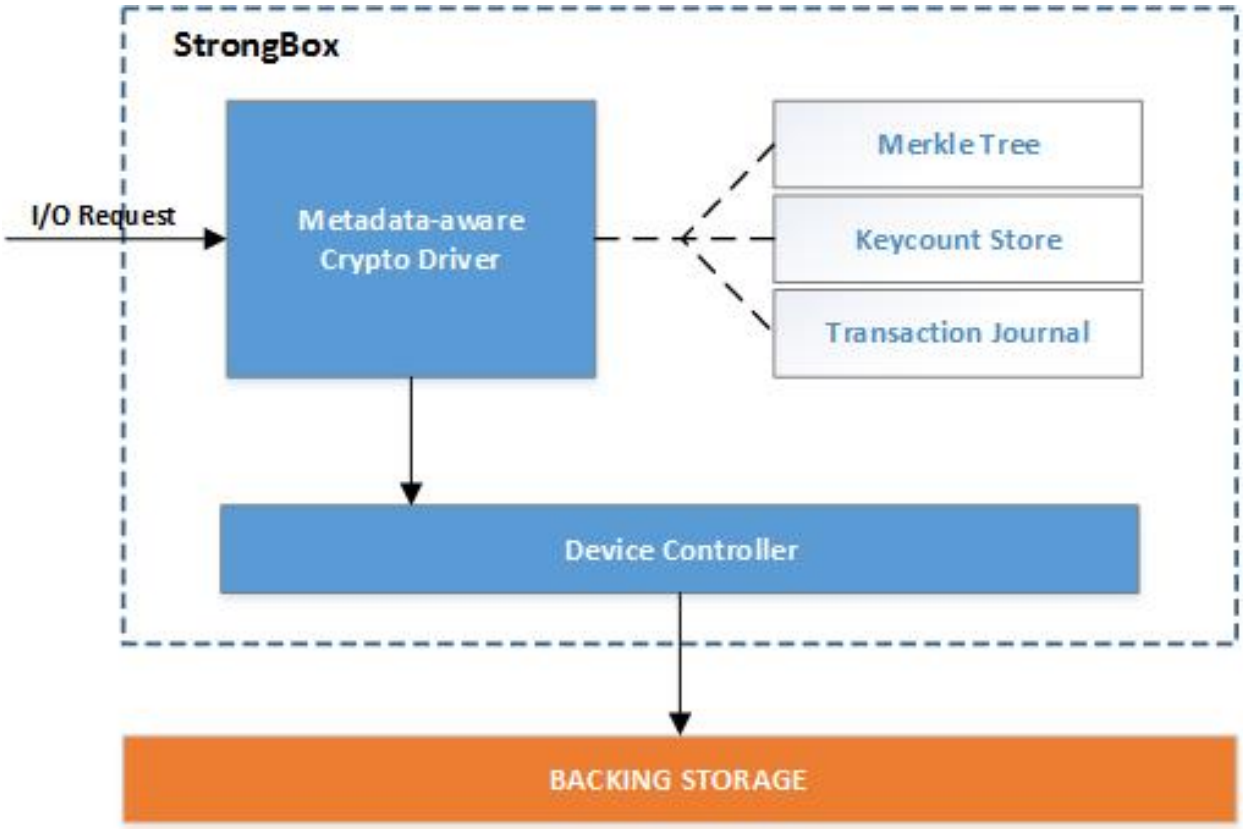


Figure 3.1: Overview of the StrongBox construction.

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. 3.2 illustrates StrongBox's layout in the backing store.

In the *Body* section of the backing store layout, end-user data is partitioned into a series of same-size logical *blocks*—distinct from the concept of physical disk blocks, which are collections of one or more disk sectors. To make this distinction clear, we refer to these wider logical blocks as *nuggets*, marked *NUG* in the Body section of Fig. 3.2. Hence, a nugget consists of one or more physical disk blocks, depending on its configured size. Each nugget is subdivided into a constant number of sub-blocks we refer to as *flakes*.

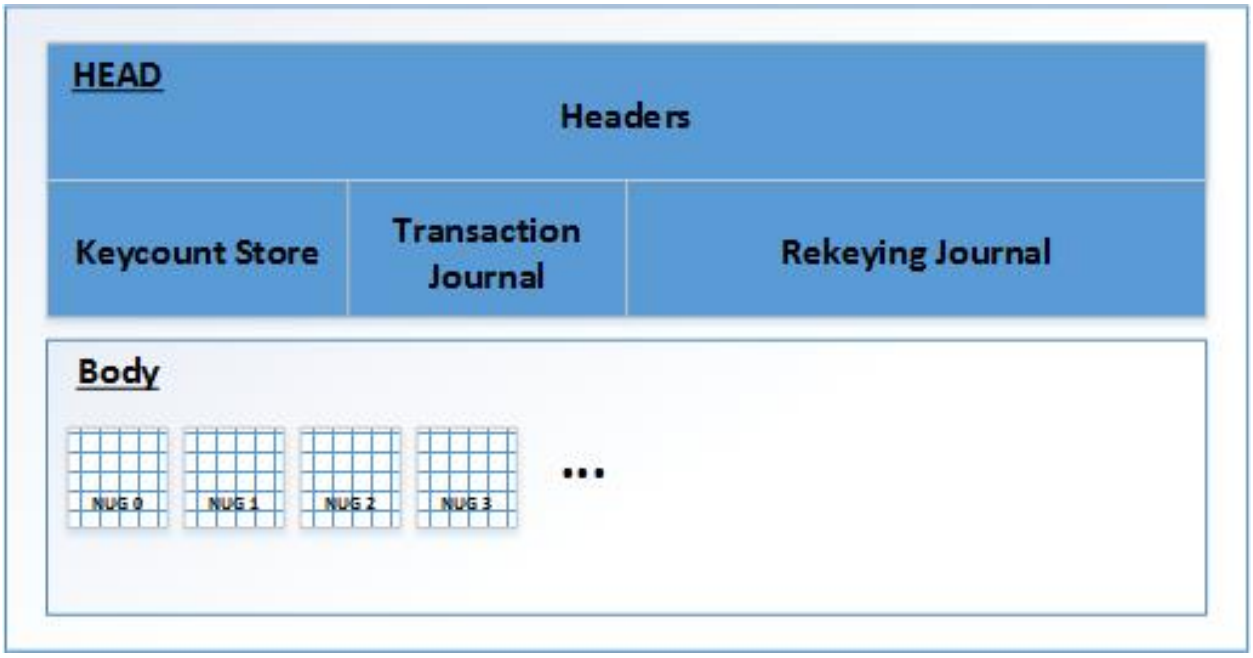


Figure 3.2: Layout of StrongBox's backing storage.

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

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

2. To track, detect, and handle overwrites in the backing store.

Hence, we must 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 "dirty". Here, dirty 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, adding to the overhead of overwrites translated by StrongBox.

The *head* section of the backing store contains the headers, which are the metadata written to disk during StrongBox initialization. These headers govern StrongBox 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 output of the Merkle Tree root
4. TPMGLOBALVER, 8 bytes; the monotonic global version count; also stored 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 per nugget
8. FLAKESIZE, 4 bytes; the size on disk 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: an array of N 8-byte integer *keycounts* and an array of N $\lceil P/8 \rceil$-byte *transaction journal entries*, where N is the number of nuggets in the backing store 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 recover to a valid state 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 with respect to affected nuggets; *i.e.,* a read that spans two nuggets would be 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 dirty in the transaction journal. If one or more of them have been marked dirty, we trigger rekeying for this specific nugget (see: Algorithm 3) and end there. Otherwise, we mark these touched flakes as dirty 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 in StrongBox breaks the security guarantee offered by any stream cipher. To prevent this failure, incoming write requests must be tracked to ensure that overlapping writes do not occur. This tracking is done with the transaction journal, featured in Fig. 3.1.

CryptedRead (Algorithm 1) 1 StrongBox in operating mode: handling an incoming read request.

Require: The read request is taken 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
1: $data \leftarrow empty$
2: **while** $\ell \neq 0$ **do**
3: $k_{n_{index}} \leftarrow GenKey_{nugget}(n_{index}, \aleph)$
4: Fetch nugget keycount n_{kc} from Keycount Store.
5: Calculate indices touched by request: f_{first}, f_{last}
6: $n_{flakedat} \leftarrow ReadFlakes(f_{first}, \ldots, f_{last})$
7: **for** $f_{current} = f_{first}$ **to** f_{last} **do**
8: $k_{f_{current}} \leftarrow GenKey_{flake}(k_{n_{index}}, f_{current}, n_{kc})$
9: $tag_{f_{current}} \leftarrow GenMac(k_{f_{current}}, n_{flakedat}[f_{current}])$
10: Verify $tag_{f_{current}}$ in Merkle Tree.
▷ (*) denotes requested subset of nugget data
11: $data \leftarrow data + Decrypt(*n_{flakedat}, k_{n_{index}}, n_{kc})$
12: $\ell \leftarrow \ell - \| * n_{flakedat} \|$
13: $n_{index} \leftarrow n_{index} + 1$
14: **return** $data$ ▷ Fulfill the read request
Ensure: $\|data\| <= \ell'$
Ensure: $\ell = 0$

The transaction journal consists of N $\lceil P/8 \rceil$-byte bit vectors where N is the number of nuggets in the backing store 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 is associated with a flake belonging to that nugget. When an incoming write request occurs that affects several flakes in a nugget, the corresponding bit vector is updated (set to 1) to reflect the new dirty 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.

CryptedWrite (Algorithm 2) 2 StrongBox in operating mode: handling an incoming write request.

Require: The write request applies 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
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 MAC algorithm and each flake's unique key to generate a per-flake MAC tag. 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.

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. 3.1 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.

Rekeying (Algorithm 3) 3 StrongBox rekeying process.

Require: The original write request applied to a contiguous segment of the backing store
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

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

▷ Iterate over all flakes in the nugget

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

3.3 Defending Against Rollbacks: Global Version Counter

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. In the case that a rollback is detected, StrongBox will refuse to initialize without warning. 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.

CHAPTER 4

STRONGBOX IMPLEMENTATION

Our implementation of StrongBox is comprised of 5000 lines of C code. Libraries used by StrongBox include OpenSSL version 1.0.2 and LibSodium version 1.0.12 for its ChaCha20, Argon2, Blake2, and AES-XTS implementations. The SHA-256 Merkle Tree implementation is borrowed from the Secure Block Device library [9]. The implementation is available at https://github.com/ananonrepo2/StrongBox.

To reduce the complexity of the experimental setup and allow StrongBox to run in user space, we provide a virtual device interface through the BUSE [1] virtual block device layer, itself based on the Network Block Device (NBD).

4.1 Deriving Subkeys

To function, the cryptographic driver must be made aware of a shared master secret. The method of derivation of this master secret is implementation specific and has no impact on performance as it is completed during StrongBox's initialization. Our implementation of StrongBox utilizes 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. Our implementation of StrongBox derives 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. Our implementation of StrongBox uses 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 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) [7]. The RPMB partition's size is fixed at manufacturing time, and all read and write commands issued to it must be authenticated by a key that is fixed when the device is first set up.

To implement rollback protection on top of the RPMB, the key for authenticating RPMB commands could be sealed in TEE sealed storage so that it would only be accessible to a specific enclave running in a TEE. This enclave would be responsible for reading and writing the StrongBox global version counter to the RPMB, and enforcing that invariant that it only increase monotonically. Our design is not dependent on the eMMC standard, however. Other trusted hardware mechanisms, including TPMs, support secure, persistent storage or monotonic counters that could be adapted for use with StrongBox.

There are two practical concerns we must address for 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 [7]. In addition, 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. 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 will almost always be substantially larger, we anticipate 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 disk sequentially and in an append-only fashion, as if it were writing to a log. This requires large amounts of contiguous space on disk, called *segments*. Since any backing store is necessarily finite in capacity, an LFS can only append so much data before it runs out of free space. When this occurs, the LFS triggers a *segment cleaning algorithm* to erase any outdated data and compress the remainder of the log down into as few segments as possible [11, 19]. This segment cleaning procedure is known more broadly as *garbage collection* [11].

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 [11]. 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 [10, 11]. 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.

CHAPTER 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, eMMC5.0 HS400 backing store) running Ubuntu Trusty 14.04 LTS, kernel version 3.10.58.

5.2 Experimental Results

In this section we seek to answer four questions:

1. What is StrongBox's overhead when compared to dm-crypt AES-XTS?
2. How does the 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?
4. How does StrongBox perform when the backing store is nearly full?

To evaluate the performance of StrongBox, 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; this data remained constant throughout the evaluation. The experiments were performed using a standard Linux ramdisk (tmpfs) as the ultimate backing store.

Ext4 ordered journaling mode (`data=ordered`) is the default mode of Ext4, where metadata is committed to the filesystem's journal while the actual data is written through to the main filesystem. In the event of a crash, the filesystem can use 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. In the event of a crash, the journal can be used to 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 was read and written per file. The median latency was 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 was dropped before every read operation, each file was opened in synchronous I/O mode via `O_SYNC`, and we relied 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 was 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.

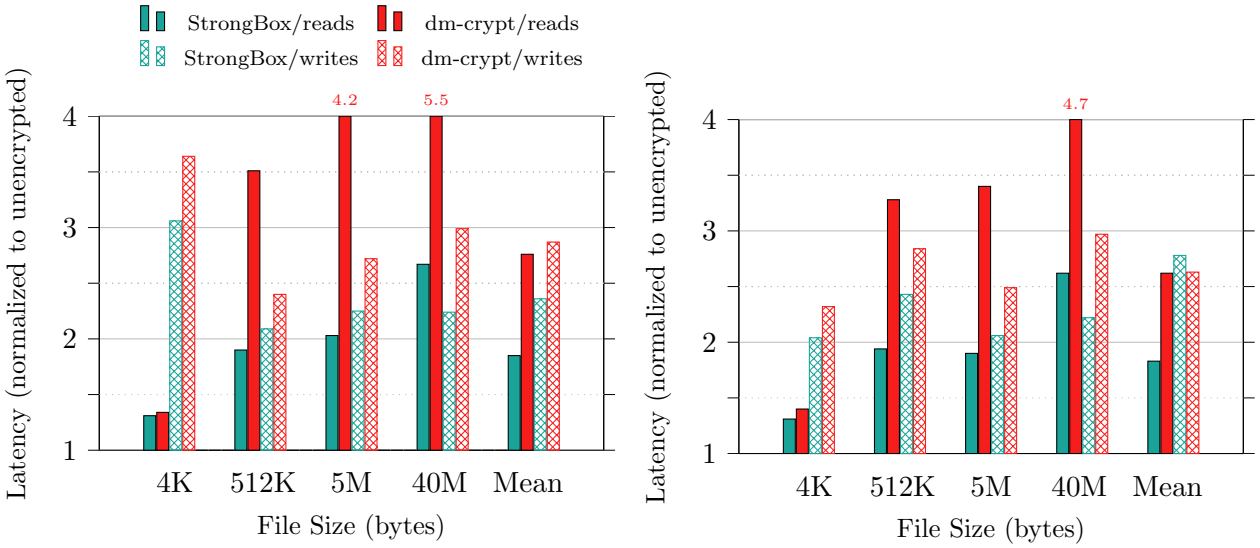
2. *StrongBox*: Filesystem mounted atop a BUSE virtual block device provided by our StrongBox implementation to perform full-disk encryption.

3. *dm-crypt*: Filesystem mounted atop a Device Mapper [17] higher-level virtual block device provided by dm-crypt to perform full-disk 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-disk encryption algorithm. All other parameters were left at their default values.

Fig. 5.1 compares StrongBox to dm-crypt under the F2FS filesystem. The gamut of result sets over different filesystems can be seen in Fig. 5.2. Fig. 5.3 compares Ext4 with dm-crypt to F2FS with StrongBox.

StrongBox vs dm-crypt AES-XTS: F2FS Test



(a) Sequential I/O expanded F2FS result set.

(b) Random I/O expanded F2FS result set.

Figure 5.1: 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 1.6× faster than dm-crypt for reads and 1.2× faster for writes.

5.3 StrongBox Read Performance

Fig. 5.1 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, 2.07× for sequential 40MB, 2.08× for sequential 5MB, 1.85× for sequential 512KB, and 1.03× for sequential 4KB.

Fig. 5.2 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× for NILFS, 2.07× for F2FS, 2.09× for Ext4 in ordered journaling mode, and 2.06× for Ext4 in full journaling mode. For smaller file sizes, the performance improvement is less pronounced. Specifically, for 4KB reads we see 1.28× for NILFS, 1.03× for F2FS, 1.07× for Ext4 in ordered journaling mode, and 1.04× 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 very slightly more performant under the NILFS LFS at 1.12×. This behavior was not observed under the more modern F2FS LFS.

5.4 StrongBox Write Performance

Fig. 5.1 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, 1.33× for sequential 40MB, 1.21× for sequential 5MB, 1.15× for sequential 512KB, and 1.19× for sequential 4KB.

Fig. 5.2 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× for F2FS and 1.18× 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. For non-LFS 4KB writes, the performance degradation is even more pronounced with a 8.09× slowdown for ordered journaling and 14.5× 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 disk (see Table 2.1). 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× improvement. For the older NILFS filesystem atop StrongBox, there is a 2.38× 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. 5.3 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 write operations, which is an instance 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 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.

5.6 Performance in StrongBox: ChaCha20 vs AES

Fig. 5.2 and Fig. 5.1 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. Fig. 5.4 describes the relationship between ChaCha20, the cipher of choice for our implementation of StrongBox, and the AES cipher. dm-crypt implements AES in XTS mode to provide full-disk encryption functionality. 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. 5.4.

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. StrongBox using OpenSSL AES-XTS experiences slowdowns of up to 1.6× for reads and 1.23× for writes across all configurations compared to StrongBox using ChaCha20. 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.

5.7 Overhead with a Full Disk

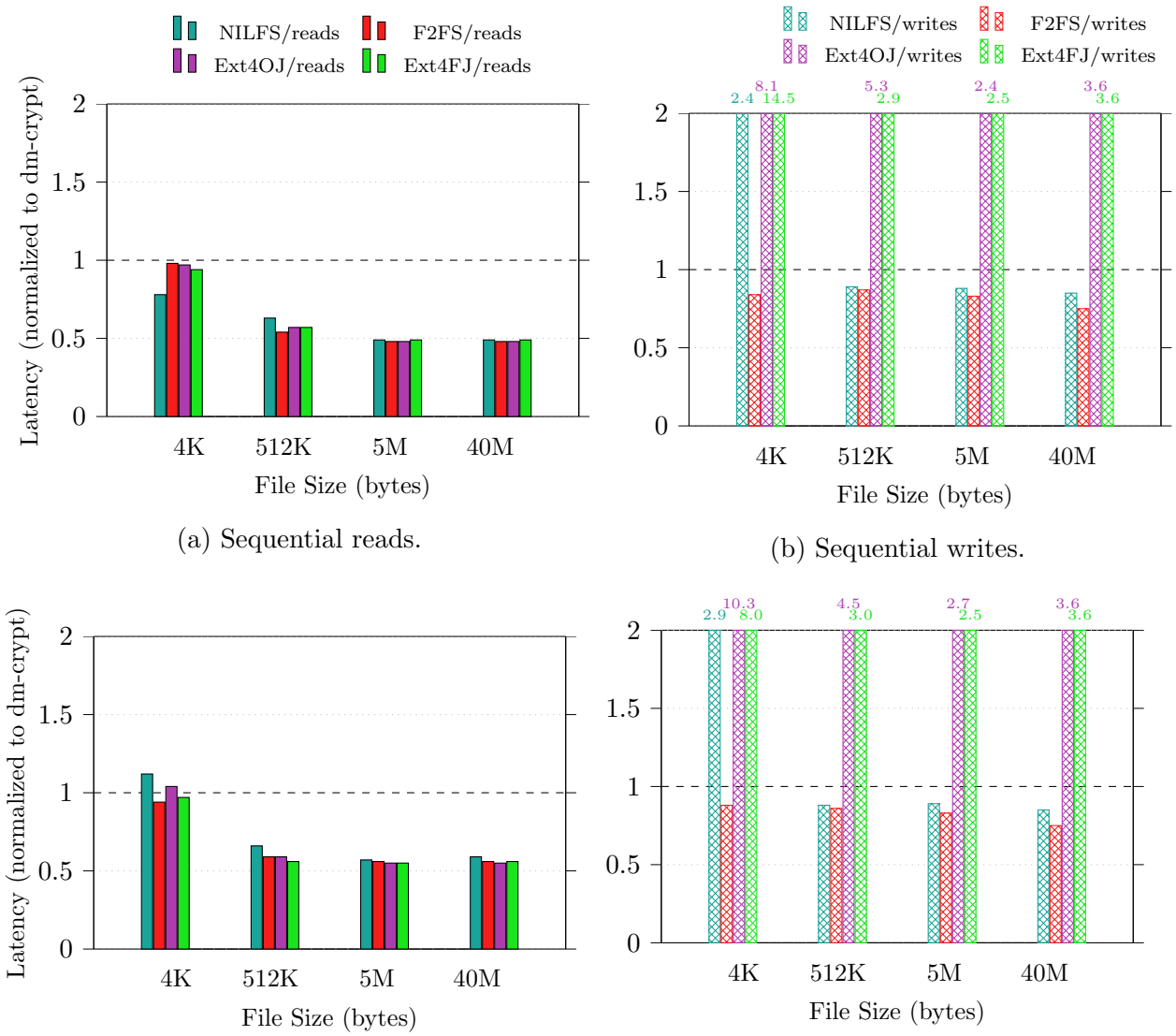
During I/O operations under an appropriate choice of LFS, we have shown that full-disk encryption provided by StrongBox outperforms full-disk encryption provided by dm-crypt. However, this is not necessarily the case when the backing store becomes full and the LFS is forced to cope with an inability 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 [11]. 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. 5.5 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 up to 3.76× as StrongBox approached maximum capacity.

Depending on the chosen garbage collection strategy (see Section 4), an LFS mounted atop a proper implementation of StrongBox would be prevented from reaching maximum capacity.



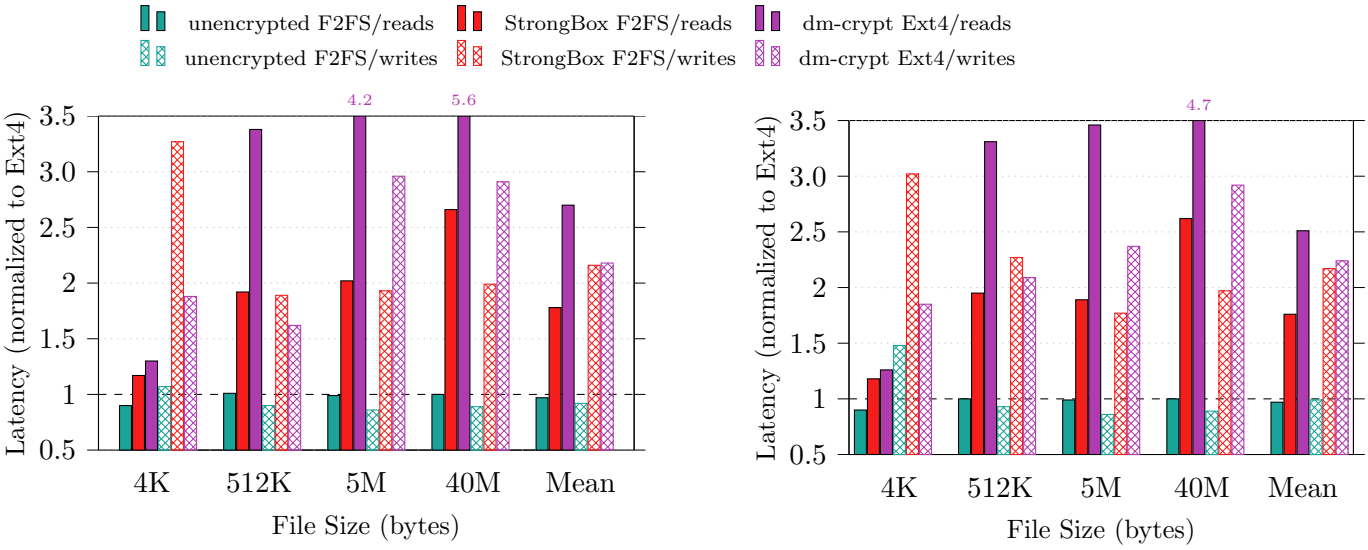
(a) Sequential reads.

(b) Sequential writes.

(c) Random reads.

(d) Random writes.

Figure 5.2: 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.



(a) Sequential I/O F2FS vs Ext4 result set.

(b) Random I/O F2FS vs Ext4 result set.

Figure 5.3: Comparison of Ext4 on dm-crypt and F2FS on StrongBox. Results are normalized to unencrypted Ext4 performance. Unecrypted F2FS results are shown for reference.

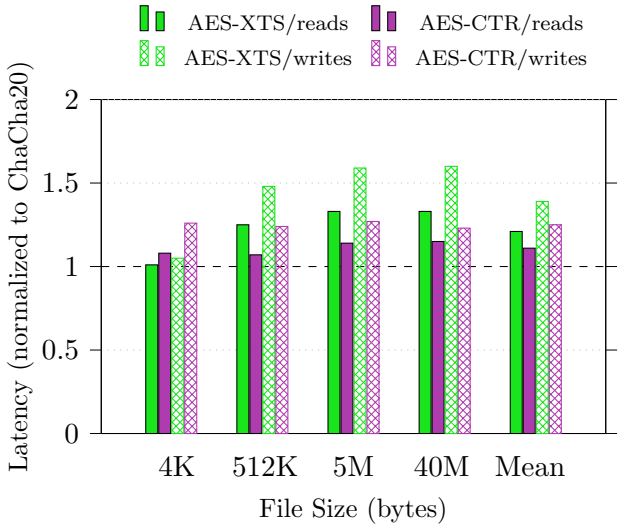


Figure 5.4: Comparison of AES in XTS and CTR modes versus ChaCha20 in StrongBox; median latency of different sized whole file sequential read and write operations normalized to ChaCha20 (default cipher in StrongBox). Points below the line signify AES outperforming ChaCha20. Points above the line signify ChaCha20 outperforming AES.

Near-Full Disk F2FS Test

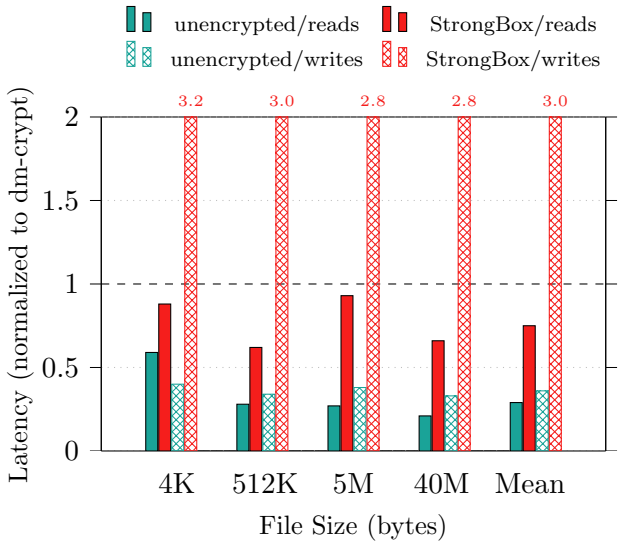


Figure 5.5: Comparison of F2FS baseline, atop dm-crypt, and atop StrongBox. All configurations are initialized with a near-full backing store; median latency of different sized whole file read and write operations normalized to dm-crypt. Points below the line are outperforming dm-crypt. Points above the line are underperforming compared to dm-crypt.

CHAPTER 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) [5, 18]. There have been numerous TES-based constructions proposed for securing data at rest [5, 8, 24], including the well known XEX-based XTS operating mode of AES [22] 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 [5, 20].

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 [2, 13]. The algorithms provided by dm-crypt all employ block ciphers [13]. Instead of a block ciphers, 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 [5, 20]. Chakraborty et al. proposed STES—a stream cipher based low cost scheme for securing stored data [5]. 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 at some level (LFS) to provide confidentiality of data.

Offering a transparent cryptographic layer at the block device level has been proposed numerous times [9]. Production implementations include storage management systems like dm-crypt. Specifically, Hein et al. proposed the Secure Block Device (SBD) [9]—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 was also implemented and evaluated using the NBD, but is not limited to one specific operating system. Further unlike StrongBox, SBD was 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 was to achieve performance parity with the industry standard AES-XTS algorithm while taking advantage of the speedup gained by utilizing a stream cipher in place of both AES and the XTS operating mode.

Achieving on-disk data integrity protection through the use of checksums has been used by filesystems and many other storage management systems. Examples include ZFS [15] and others [9]. For our implementation of StrongBox, we used the Merkle Tree library offered by SBD to manage our in- memory checksum verification. An implementation of StrongBox need not use the SDB SHA-256 Merkle Tree library. It was chosen for convenience.

CHAPTER 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: the 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 ChaCha20 and 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 1.21× improvement on write performance. In fact, our results show that F2FS plus StrongBox provides a higher performance replacement for Ext4 backed with dm-crypt. We have made our implementation of StrongBox available as open source so that others can extend it or compare to it. Our hope is that this work motivates further exploration of fast stream ciphers as replacements for AES-XTS for securing data at rest.

CHAPTER 8

FUTURE WORK

8.1 Integrate StrongBox Into F2FS

To provide full-disk encryption, StrongBox was designed to offer a transparent cryptographic layer at the block device level. Necessarily, no specialized file systems nor kernel interface changes are required to make use of StrongBox. As made evident by this research, the transparency of StrongBox is a significant source of overhead. Specifically, it is necessary for StrongBox to maintain expensive metadata structures within the cryptographic driver to prevent fatal overwrites by the overlying file system. These structures include the in-memory *Merkle Tree* and disk-backed *Transaction Journal*, which are checked on every I/O operation and updated on every write operation as well as the *Keycount Store*, which is mutated during StrongBox's rekeying procedure after an overwrite has been detected.

For F2FS, this metadata maintenance results in slowdowns of up to 3× compared to unencrypted I/O (see Fig. 5.1). This blanket expense is made more egregious by the fact that F2FS very rarely commits overwrites (see Table 2.1), making this metadata truly useful in only a few instances when F2FS has not triggered garbage collection.

Instead of providing a transparent cryptographic layer below F2FS, we can conceivably do away with a significant chunk of StrongBox's metadata management responsibilities by integrating the now-redundant Transaction Journal into F2FS's own internal metadata structure, namely the Checkpoint (CP) and Segment Information Table (SIT) structures which already maintain validity bitmaps for segments and their main area blocks. This would provide the same functionality as a distinct StrongBox under F2FS, but with the performance benefit that comes with shaving down on metadata maintenance.

We can take this a step further when we consider overwrite detection and correction. With StrongBox integrated directly into F2FS, I/O operations can be scheduled so as to

control the location and reduce the frequency of costly overwrites, perhaps even eliminating them altogether outside of garbage collection.

8.2 Explore the Trade-off Between Energy, Performance, and Security

StrongBox utilizes ChaCha20 as its stream cipher of choice. ChaCha20 is not the only eligible stream cipher (see Section 3), nor is it the fastest in the ChaCha family of stream ciphers [4]. Indeed, the twenty round "ChaCha20" is the *slowest* of the available ChaCha implementations, which include an eight and twelve round "ChaCha8" and "ChaCha12" respectively. We selected ChaCha20 for use with StrongBox because the ChaCha standard considers 20 rounds, but there is to our knowledge no cryptanalytic threat towards the ChaCha8/12 implementations [14]. Other interesting stream ciphers include Rabbit, Trivium, and even AES in CTR mode.

In exchange for a technically looser security guarantee depending on the selected stream cipher, there may be a navigable trade-off space to exploit for improved performance and/or energy/power savings under certain conditions. Such a trade-off space could be explored using our implementation of StrongBox with minimal API changes. StrongBox's modular design lends itself well to the dynamic swapping of algorithms at runtime.

8.3 Investigate ChaCha20 Energy Usage

In respect to I/O operation latency, StrongBox outperforms its counterpart dm-crypt with AES-XTS in the majority of cases. In respect to energy and power use per I/O operation, however, the matter becomes more interesting. Though it was not explored in this research, we note that StrongBox's energy use was often erratic compared to dm-crypt and is deserving of further study.

Specifically, there are several StrongBox configurations where we see a reduction in I/O operation latency over dm-crypt accompanied by a proportional increase in power consumption by StrongBox. Though energy usage is outside of the scope of this research, this presents an immediate area for further inquiry.

BIBLIOGRAPHY

[1] *A block device in userspace*. 2012. URL: https://github.com/acozzette/BUSE (visited on 04/26/2017).

[2] *Android Open Source Project: Full-Disk Encryption*. URL: https://source.android.com/security/encryption/full-disk (visited on 04/26/2017).

[3] D. J. Bernstein. *The Poly1305-AES message-authentication code*. Tech. rep. University of Illinois at Chicago, 2005.

[4] D. J. Bernstein. *ChaCha, a variant of Salsa20*. Tech. rep. University of Illinois at Chicago, 2008.

[5] D. Chakraborty, C. Mancillas-Lpez, and P. Sarkar. "STES: A Stream Cipher Based Low Cost Scheme for Securing Stored Data". In: *IEEE Transactions on Computers* 64.9 (2015), pp. 2691–2707. ISSN: 0018-9340. DOI: 10.1109/TC.2014.2366739.

[6] M. Cornwell. "Anatomy of a Solid-state Drive". In: *Queue* 10.10 (Oct. 2012), 30:30–30:36. ISSN: 1542-7730. DOI: 10.1145/2381996.2385276. URL: http://doi.acm.org/10.1145/2381996.2385276.

[7] *EMBEDDED MULTI-MEDIA CARD (eMMC), ELECTRICAL STANDARD (5.1)*. 2015. URL: https://www.jedec.org/standards-documents/results/jesd84-b51 (visited on 04/26/2017).

[8] S. Halevi and P. Rogaway. "A Tweakable Enciphering Mode". In: *Advances in Cryptology - CRYPTO 2003: 23rd Annual International Cryptology Conference, Santa Barbara, California, USA, August 17-21, 2003. Proceedings*. Ed. by D. Boneh. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 482–499. ISBN: 978-3-540-45146-4. DOI: 10.1007/978-3-540-45146-4_28. URL: http://dx.doi.org/10.1007/978-3-540-45146-4_28.

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

[10] R. Konishi, Y. Amagai, K. Sato, H. Hifumi, S. Kihara, and S. Moriai. "The Linux Implementation of a Log-structured File System". In: *SIGOPS Oper. Syst. Rev.* 40.3 (July 2006), pp. 102–107. ISSN: 0163-5980. DOI: 10.1145/1151374.1151375. URL: http://doi.acm.org/10.1145/1151374.1151375.

[11] C. Lee, D. Sim, J. Hwang, and S. Cho. "F2FS: A New File System for Flash Storage". In: *13th USENIX Conference on File and Storage Technologies (FAST 15)*. Santa Clara, CA: USENIX Association, 2015, pp. 273–286. ISBN: 978-1-931971-201. URL: https://www.usenix.org/conference/fast15/technical-sessions/presentation/lee.

[12] A. Limited. *ARM security technology: Building a secure system using TrustZone technology*. PRD29-GENC-009492C. 2009.

[13] *Linux kernel device-mapper crypto target*. 2013. URL: https://gitlab.com/cryptsetup/cryptsetup (visited on 04/26/2017).

[14] S. Maitra. *Chosen IV Cryptanalysis on Reduced Round ChaCha and Salsa*. Tech. rep. Applied Statistics Unit, Indian Statistical Institute, 2015.

[15] *Oracle blog: ZFS End-to-End Data Integrity*. 2005. URL: https://blogs.oracle.com/bonwick/zfs-end-to-end-data-integrity (visited on 04/26/2017).

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

[17] *RedHat: Device-mapper Resource Page*. URL: https://www.sourceware.org/dm (visited on 04/26/2017).

[18] P. Rogaway. *Efficient Instantiations of Tweakable Blockciphers and Refinements to Modes OCB and PMAC*. Tech. rep. University of California at Davis, 2004.

[19] M. Rosenblum and J. K. Ousterhout. "The Design and Implementation of a Log-structured File System". In: *ACM Trans. Comput. Syst.* 10.1 (Feb. 1992), pp. 26–52. ISSN: 0734-2071. DOI: 10.1145/146941.146943. URL: http://doi.acm.org/10.1145/146941.146943.

[20] P. Sarkar. *Tweakable Enciphering Schemes From Stream Ciphers With IV*. Tech. rep. Indian Statistical Institute, 2009.

[21] G. P. D. Technology. *TEE client API specification version 1.0*. GPD_SPE_007. 2010.

[22] *The XTS-AES Tweakable Block Cipher*. IEEE Std 1619-2007. 2008.

[23] *TLS Symmetric Crypto*. 2014. URL: https://www.imperialviolet.org/2014/02/27/tlssymmetriccrypto.html (visited on 04/26/2017).

[24] P. Wang, D. Feng, and W. Wu. "HCTR: A Variable-Input-Length Enciphering Mode". In: *Information Security and Cryptology: First SKLOIS Conference, CISC 2005, Beijing, China, December 15-17, 2005. Proceedings*. Ed. by D. Feng, D. Lin, and M. Yung. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 175–188. ISBN: 978-3-540-32424-9. DOI: 10.1007/11599548_15. URL: http://dx.doi.org/10.1007/11599548_15.