SwitchCrypt: Navigating Tradeoffs in Stream Cipher Based Full Drive Encryption

Abstract

Recent work on Full Drive Encryption shows that stream ciphers achieve significantly improved performance over block ciphers while offering stronger security guarantees. However, optimizing for performance often conflicts with other key concerns like energy usage and desired security properties. In this paper we present SwitchCrypt, a software mechanism that navigates the tradeoff space made by balancing competing security and latency requirements via cipher switching *in space or time. Our key insight in achieving low-overhead switching is to leverage the overwrite-averse, append-mostly behavior of underlying solid-state storage to trade throughput for reduced energy use and/or certain security properties. We implement SwitchCrypt on an ARM big.LITTLE mobile processor and test its performance under the popular F2FS file system. We provide empirical results demonstrating the conditions under which different switching strategies are optimal through the exploration of four case studies. In one study, where we require the filesystem to react to a shrinking energy budget by switching ciphers, we find that SwitchCrypt achieves up to a 3.3x total energy use reduction compared to a static approach using only the Freestyle stream cipher. In another case, where we allow the user to manually switch between ChaCha20 and Freestyle stream ciphers dynamically, we achieve a 1.6x to 4.8x reduction in I/O latency compared to prior static approaches.*

1. Introduction

The state of the art for securing data at rest, such as the contents of a laptop’s SSD, is Full Drive Encryption (FDE). Traditional FDE employs a single cryptographic cipher to encrypt, decrypt, and authenticate drive contents on the fly. Unfortunately, encryption introduces overhead that drastically impacts system performance and total energy consumption. Hence, modern storage systems must carefully balance these competing concerns. On the one hand, stronger security guarantees come with increased latency and threaten to balloon energy consumption. On the other, capping total energy consumption requires tolerating increased latency or weaker security guarantees.

FDE implementations such as dm-crypt [9, 2] for Linux and BitLocker for Windows [**?**] balance these concerns with respect to some generic “common case” workload, but this approach often fails to deliver an optimal outcome for specific workloads. Google’s Android OS is a good example. Android supported FDE with the release of Android 3.0, yet it was not enabled by default until Android 6.0 [16]. Two years prior, Google attempted to roll out FDE by default on Android 5.0 but had to backtrack. In a statement to Engadget, Google blamed “’performance issues on some partner devices’ ... for the backtracking” [31]. At the same time, AnandTech reported a “62.9% drop in random read performance, a 50.5% drop in random write performance, and a staggering 80.7% drop in sequential read performance” versus Android 5.0 unencrypted storage for various workloads [23].

FDE’s impact on drive performance and energy efficiency depends on a multitude of choices. Paramount among them is the choice of cipher, as different ciphers expose different performance and energy efficiency characteristics. In this way, cipher choice can be viewed as a key *configuration parameter* for FDE systems. The benefits of choosing one cipher over another might include: 1) improved performance (*i.e.,* overall reduction in FDE overhead), 2) reduced energy use, 3) more useful security guarantees, and 4) the ability to encrypt devices that would otherwise be too slow or energy-inefficient to support FDE.

However, the standard cipher choice for FDE is the slow AES *block cipher* (AES-XTS) [6, 29, 7]. Dickens et al. introduced a method for using ciphers other than AES-XTS for FDE; specifically, the high performance ChaCha20 *stream cipher* [17, 12]. More recent work—such as Google’s HBSH (hash, block cipher, stream cipher, hash) and Adiantum [15]—brings stream cipher based FDE to devices that do not or cannot support hardware accelerated AES. Hence, it has been demonstrated that using stream ciphers for FDE is both desirable in a variety of contexts and feasible at industry scale.

In this paper, we explore the rich configuration space of stream ciphers beyond the narrow scope of previous work. Each cipher comes with a variety of performance, energy efficiency, and security properties in the FDE context. These ciphers include ChaCha variants with different round counts (e.g., ChaCha8 [] and ChaCha12) [12], ciphers with stronger security guarantees versus more robust adversaries (e.g., Freestyle [10]), and other ciphers like SalsaX [13], AES in counter mode (AES-CTR) [4], Rabbit [14], Sosemanuk [11], etc. Given this variety, the cipher configuration providing the least latency overhead is almost always different than the configuration providing the most desirable security properties, and both may differ from the configuration using the least energy or preserving the most free space on the encrypted drive.

Further, while cipher choice might be configured statically at compile or boot time with respect to some common case in traditional FDE, such configurations cannot dynamically adapt to changes that arise while the system is running. Examples include changes in resource availability, runtime environment, desired security properties, and respecting changing OS energy budgets.

Hence, any static common-case FDE configuration will sacrifice one concern for another, even when it is not optimal to do so for a given workload. But what if a system could dynamically transition into a more desirable configuration given runtime changes?

Our Contributions

To realize the goal of dynamically adjusting storage tradeoffs at runtime, we:

- Define a scheme to *quantify* the usefulness of each cipher based on key security properties relevant to FDE in context. Using this scheme, we define a tradeoff space of cipher configurations over competing concerns: total energy use, desirable security properties, read and write performance (latency), total writable space on the drive, and how quickly the contents of the drive can converge to a single encryption configuration.
- Introduce a novel design substantively expanding prior FDE work by wholly *decoupling* cipher implementations from the encryption process.
- Develop the idea of cipher switching using *switching strategies* to "re-cipher" storage units dynamically, allowing us to tradeoff different performance and security properties of various configurations at runtime.

Unifying these contributions, we present SwitchCrypt: a software system that navigates the tradeoff space made by balancing competing security and latency requirements via cipher switching in space and time. We implement SwitchCrypt and three switching strategies—*Forward*, *Selective*, and *Mirrored*—to dynamically transition the system between configurations using our generic cipher interface. We then study the utility of cipher switching through four case studies where latency, energy, and desired security properties change over time.

In one study, where we require the filesystem to react to a shrinking energy budget by switching ciphers, we find that SwitchCrypt achieves up to a 3.3x total energy use reduction compared to prior static approaches that only use the Freestyle stream cipher without switching. In another case, where we allow the user to manually switch between ChaCha20 and Freestyle stream ciphers dynamically, we achieve a 3.1x to 4.8x reduction to read latency and 1.6x to 2.8x reduction to write latency compared to prior static approaches.

We make the SwitchCrypt source publicly available open source[1].

2. Motivation

2.1. Example: Filesystem Reacts to "Battery Saver"

Suppose we have an ARM-based ultra-low-voltage netbook provided to us by our employer. As this is an enterprise device, our employer requires that 1) our drive is fully encrypted at all times and 2) our encrypted data is constantly backed up to an offsite system. The industry standard in full drive encryption is AES-XTS, so we initialize our drive with it. Given these requirements, three primary concerns present themselves.

First, it is well known that FDE using AES-XTS adds significant latency and power overhead to I/O operations, especially on mobile and battery-constrained devices [31, 16, 23]. To keep our drive encrypted at all times with AES-XTS means we must accept this hit to performance and battery life. Worse, if our device does not support hardware accelerated AES, performance can be degraded even further; I/O latency can be as high as 3–5x [17]. Hardware accelerated AES is hardly ubiquitous, and the existence of myriad devices that do not support it cannot simply be ignored, hence Google's investment in Adiantum—an FDE solution for "the next billion users" of low-cost devices without support for hardware accelerated AES [15].

Second, AES-XTS is designed to mitigate threats to drive data "at rest," which assumes an attacker cannot access snapshots of our encrypted data nor manipulate our data without those manipulations being immediately obvious to us. With access to multiple snapshots of a drive's AES-XTS-encrypted contents, an attacker can passively glean information about the plaintext over time by contrasting those snapshots, leading to confidentiality violations in some situations [30, 6]. Similarly, an attacker that can manipulate encrypted bits without drawing our attention will corrupt any eventual plaintext, violating data integrity and, in the worst case, influencing the behavior of software. Unfortunately, in real life, data rarely remains "at rest" in these ways. In our example, our employer requires we back up the contents of our drive to some offsite backup service; this service will receive periodic snapshots of the encrypted state of our drive, violating our "at rest" invariant. These backups occur at a layer above the drive controller, meaning any encryption happening at the FTL or below is irrelevant.

Third, our system is battery constrained, placing a cap on our energy budget that can change at any moment as we transition from line power to battery power and back. Our system should respond to these changing requirements without violating any other concerns.

To alleviate the performance concern, we can choose a stream cipher like ChaCha20 rather than the AES-XTS block cipher. Using StrongBox, an encryption driver built for ChaCha20-based FDE, we can achieve on average a 1.7x speedup and a commensurate reduction in energy use [17].

When it comes to the security concern, StrongBox solves both the snapshot and integrity problems by 1) never writing data encrypted with the same key to the same location and 2) tracking drive state using a Merkle tree and monotonic counter supported by trusted hardware to prevent rollbacks. This ensures data manipulations cannot occur and guarantees confidentiality even when snapshots are compared regardless

[1] https://github.com/ananonrepo2/SwitchCrypt

of the stream cipher used. Unfortunately, restoring from a backup necessitates a forced rollback of drive state, potentially opening us back up to confidentiality-violating snapshot comparison attacks [17].

To truly address the security concern requires a cipher with an additional security property: *ciphertext randomization*. Without ciphertext randomization, an attacker can map plaintexts to their ciphertext counterparts during snapshot comparison, especially if they can predict what might be written to certain drive regions. However, with ciphertext randomization, a cipher will output a different "random" ciphertext even when given the same key, nonce, and plaintext; this means, even after a forced rollback of system state and/or legitimate restoration from a backup, comparing future writes is no longer violates confidentiality because each snapshot will always consist of different ciphertext regardless of the plaintext being encrypted or the state of the drive. Using Freestyle [10], a ChaCha20-based stream cipher that supports ciphertext randomization, we can guarantee data confidentiality in this way. So, we switch from StrongBox to an encryption driver that supports Freestyle.

Unfortunately, like AES-XTS, Freestyle has significant overhead compared to the original ChaCha20. In exchange for stronger security properties, Freestyle is up to 1.6x slower than ChaCha20, uses more energy, has a higher initialization cost, and expands the ciphertext which reduces total writeable drive space [10].

Further complicating matters is our final concern: a constrained energy budget. Our example system is battery constrained. Even if we accepted trading off performance, drive space, and energy for security in some situations, in other situations we might prioritize reducing total energy use. For example, when we trigger "battery saver" mode, we expect our device to conserve as much energy as possible. It would be ideal if our device could pause backups and the encryption driver could switch from the ciphertext-randomizing Freestyle configuration back to our high performance energy-efficient ChaCha20 configuration when conserving energy is a top priority, and then switch back to the Freestyle configuration when we connect to a charger and backups are eventually resumed.

In this paper we present SwitchCrypt, a device mapper that can trade off between these two configurations and others without compromising security or performance or requiring the device be restarted. With prior work, the user must select a static operating point in the energy-security-latency space at initialization time and hope it is optimal across all workloads and cases. If they choose the Freestyle configuration, their device will be slower and battery hungry even when they are not backing up. If they choose the more performant ChaCha20 configuration, they risk confidentiality-violating snapshot comparison attacks. SwitchCrypt solves this problem by encrypting data with the high performance ChaCha20 when the battery is low and switching to Freestyle when plugged in and syncing with the backup service resumes.

2.2. Key Challenges

To trade off between different cipher configurations, we must address three key challenges. First, we must determine what cipher configurations are most desirable in which contexts and why. This requires we *quantify* the desirable properties of these configurations. Second, we must have some way to encrypt independent storage units with any one of these configurations. This requires we *decouple* cipher implementations from the encryption process used in prior work. Third, we need to determine when to re-encrypt those units, which configuration to use, and where to store the output, all with minimal overhead. This requires we implement efficient cipher *switching strategies*.

Quantifying the properties traded off between configurations. To obtain a space of configurations that we might reason about, it is necessary to compare certain properties of stream ciphers useful in the FDE context. However, different ciphers have a wide range of security properties, performance profiles, and output characteristics, including those that randomize their outputs and those with non-length-preserving outputs—*i.e.,* the cipher outputs more data than it takes in. To address this, we propose a framework for quantitative cipher comparison in the FDE context; we use this framework to define our configurations.

Decoupling ciphers from the encryption process. To flexibly switch between configurations in SwitchCrypt requires a generic cipher interface. This is challenging given the variety of inputs required by various stream ciphers, the existence of non-length-preserving ciphers, and other differences. We achieve the required generality by defining independent storage units called *nuggets*; we borrow this terminology from prior work (see [17]) to easily differentiate our logical blocks (nuggets) from physical drive and other storage blocks. And since they are independent, we can use our interface to select any configuration to encrypt or decrypt any nugget at any point.

Implementing efficient switching strategies. Finally, to determine when to switch a nugget's cipher and to where we commit the output, we implement a series of high-level policies we call *cipher switching strategies*. These strategies leverage our generic cipher interface and flexible drive layout to selectively "re-cipher" groups of nuggets, whereby the key and the cipher used to encrypt/decrypt a nugget are switched at runtime. These strategies allow SwitchCrypt to move from one configuration point to another or even settle on optimal configurations wholly unachievable with prior work. The challenge here is to accomplish this while minimizing overhead.

3. SwitchCrypt Design

3.1. SwitchCrypt Overview

SwitchCrypt consists of a *Generic Stream Cipher Interface* and *Cryptographic Driver*; SwitchCrypt sits between a Log-

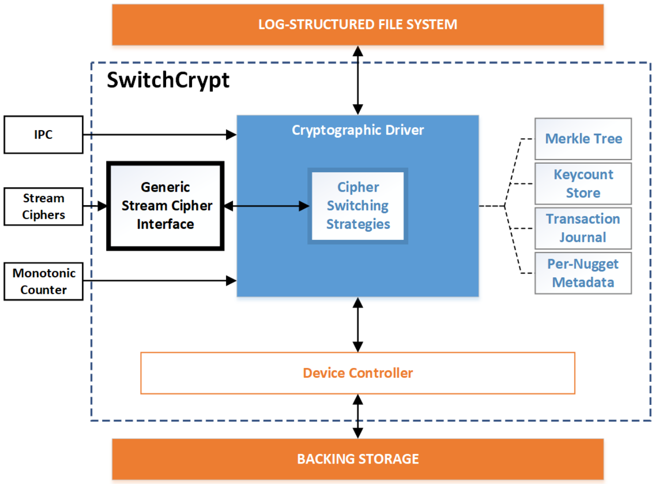

Figure 1: Overview of the SwitchCrypt construction.

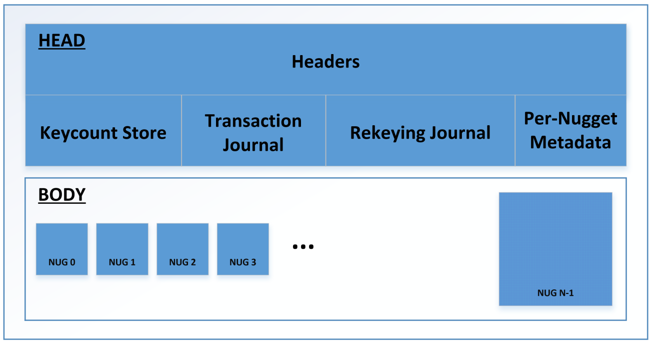

Figure 2: Layout of SwitchCrypt's drive layout.

structured File System (LFS) on the OS, and the underlying drive (backing storage) and device controller (e.g. Flash Translation Layer). This is illustrated in Fig. 1, which provides an overview of the SwitchCrypt system design.

The drive itself is divided into a *HEAD* section and *BODY* section upon initialization, illustrated in Fig. 2. The HEAD consists of metadata headers written during initialization [17] along with the *Keycount Store*, *Transaction Journal*, *Rekeying Journal*, and *Per-Nugget Metadata*, each drive-backed. These components are used by the *Cryptographic Driver* together with the *Cipher Switching Strategy* implementations to enable efficient per-unit cipher switching.

The BODY consists of a series independent same-size logical units called *nuggets*. A nugget consists of one or more contiguous physical drive blocks. Each nugget is coupled with metadata in the HEAD indicating which cipher was used to encrypt the nugget along with any additional ciphertext output; the latter allows us to treat any non-length-preserving ciphers as if they were length-preserving. SwitchCrypt uses the Keycount Store and Transaction Journal components along with our nugget layout to 1) track, detect, and handle overwrites, 2) limit the maximum length of any plaintext input to ciphers, thus amortizing the overhead incurred during encryption, and 3) independently and efficiently switch the cipher used to encrypt individual nuggets.

Dickens et al. showed how to make nugget-based drive organization secure using a single stream cipher, ChaCha20, to handle overwrites, prevent rollback attacks, and limit plaintext length [17]. However, they did not envision the utility in trading off between concerns at the filesystem or device-mapper level, dynamic cipher switching, or protecting against attacks on data "in motion;" the remainder of this section details the novel components that enable this functionality. Specifically: how we quantify the properties traded off between configurations (§3.2), the Generic Stream Cipher Interface and Per-Nugget Metadata components (§3.3) which decouple cipher implementations from the encryption process, and our Cipher Switching Strategy implementations (§3.4) used to efficiently encrypt nuggets with different ciphers.

3.2. Quantifying Cipher Security Properties

To reason about when to trade off between the ciphers evaluated in this work, we must have a way to compare ciphers' utility in the context of SwitchCrypt FDE. However, different ciphers have a wide range of security properties, performance profiles, and output characteristics. To address this need, we propose a novel evaluation framework (see: Table 1). Our framework classifies stream ciphers according to three quantitative features: relative round count, ciphertext randomization, and ciphertext expansion. Taken together, these features reveal a rich tradeoff space of cipher configurations optimizing for different combinations of concerns.

Cipher	Rounds	Randomization	Expansion
ChaCha8	0	0	1
ChaCha12	0.5	0	1
ChaCha20	1	0	1
Salsa8	0	0	1
Salsa12	0.5	0	1
Salsa20	1	0	1
HC128	0	0	1
HC256	1	0	1
Freestyle (F)	0	2	0
Freestyle (B)	0.5	2.5	0
Freestyle (S)	1	3	0

Table 1: Our framework for classifying stream ciphers according to three ideal features: relative round count, ciphertext randomization, and ciphertext expansion.

3.2.1. Relative Rounds (Rounds) The ciphers we examine in this paper are all constructed around the notion of *rounds*, where a higher number of rounds (and possibly longer key) is positively correlated with a higher resistance to brute force given no fatal related-key or other attacks [26]. Hence, this feature represents how many rounds a cipher executes relative to other implementations of the same algorithm. For instance: ChaCha8 is a reduced-round version of ChaCha12, which is a reduced-round version of ChaCha20, all using the ChaCha algorithm [12, 26].

We limit our analysis to groups of three implementations, each using a different number of rounds. In the case of HC-128 and HC-256, we limit our analysis to a group of two implementations. Scores range from 0 (least number of rounds considered) to 1 (greatest number of rounds considered).

3.2.2. Ciphertext Randomization (Randomization) A cipher with ciphertext randomization generates different ciphertexts non-deterministically given the same key, nonce, and plaintext. This makes it much more difficult to execute chosen-ciphertext attacks (CCA), key re-installation attacks, XOR-based cryptanalysis and other comparison attacks, and other confidentiality-violating schemes where the ciphertext is in full control of the adversary [10]. This property is useful in cases where we cannot prevent the same key, nonce, and plaintext from being reused, such as with data "in motion" (see the motivational example in Section 2). Ciphers without this property—such as ChaCha20 on which prior work is based—are trivially broken when key-nonce-plaintext 3-tuples are reused. In StrongBox, this is referred to as an "overwrite condition" or simply "overwrite" [17].

Though there are many ways to achieve ciphertext randomization, the ciphers included in our analysis implement it using a random number of rounds for each block of the message where the exact number of rounds are unknown to the receiver a priori [10]. In determining the minimum and maximum number of rounds used per block in this non-deterministic mode of operation, we can customize the computational burden an attacker must bear by choosing lower or higher minimums and maximums. Hence, this is not a binary feature; scores range from 0 (no ciphertext randomization) to 1 (lowest minimum and maximum rounds per block) to 3 (highest minimum and maximum rounds per block).

3.2.3. Ciphertext Expansion (Expansion) A cipher that exhibits ciphertext expansion is non-length-preserving: it outputs more or less ciphertext than was originally input as plaintext. This can cause major problems in the FDE context. For instance, cryptosystems that rely on AES-XTS (e.g. Linux's dm-crypt, Microsoft's BitLocker, Apple's FileVault) or ChaCha (e.g. StrongBox, Google's Adiantum) have storage layouts that hold length-preserving output as an invariant, making ciphers that do not exhibit this property incompatible with their implementations; yet, ciphertext expansion is often (but not always) a necessary side-effect of ciphertext randomization.

The ciphers included in our analysis that exhibit ciphertext expansion have an overhead of around 1.56% per plaintext message block [10]. Even a single byte of additional ciphertext vs plaintext would make a cipher inappropriate for use with prior work. Hence, this is a binary feature in that a cipher either outputs ciphertext of the same length as its plaintext input or it does not. A cipher scores either a 0 if it *is not* length-preserving in this way or a 1 if the ciphertext is always the same length as the plaintext.

3.3. Generic Stream Cipher Interface

One of the goals of SwitchCrypt is that we might use any stream cipher regardless of its implementation details. Yet this is entirely non-trivial. There are many cipher implementations that we might use with SwitchCrypt, each with unique input requirements and output considerations. For instance, Salsa and Chacha implementations require a certain IV and key size and handle plaintext input through successive invocations of a single state update function [18]. Using OpenSSL's AES implementation in CTR mode requires manually tracking the counter state and individual ciphertext blocks are retrieved though corresponding function invocations [**?**]. Freestyle's reference implementation requires we calculate the extra space necessary per nugget (due to ciphertext expansion) along with configuration-dependent minimum and maximum rounds-per-block, hash interval, and pepper bits [10]. HC-128 and other ciphers have similarly disparate requirements.

Unlike prior work, SwitchCrypt must be able to encrypt and decrypt arbitrary nuggets *with any of these ciphers* at any moment with low overhead and without tight coupling to any specific implementation detail. Hence, there is a need for an interface that completely decouples cipher implementations from the encryption/decryption process. Our novel cipher interface allows any stream cipher to be integrated into SwitchCrypt without modifications to third-party code, enabling normally incompatible ciphers to encrypt and decrypt arbitrary nuggets. The ability for disparate cipher implementations to co-exist forms the foundation for SwitchCrypt's ability to switch the system between different cipher configurations in our tradeoff space efficiently and effectively.

To facilitate this, the Generic Stream Cipher Interface presents the cryptographic driver with a single unified encryption/decryption model. SwitchCrypt receives I/O requests from the operating system at the block device level like any other device-mapper. These requests come in the form of either reads or writes. When a read request is received, the OS hands SwitchCrypt an offset and a length and expects a response with plaintext of that specific length starting at that specific offset taken from the beginning of storage (i.e. the BODY section; see Fig. 2). When a write request is received, the OS hands SwitchCrypt an offset, a length, and a buffer of plaintext and expects that plaintext to be encrypted and committed to storage such that the plaintext is later retrievable given that same offset and length in a future read request. These requests can either be handled together by a single function or handled individually as distinct read and write operations, each with different tradeoffs.

1. **`xor_interface`**
   `xor_interface` executes independently of SwitchCrypt internals and treats encryption and decryption as the same operation. Implementations receive an integer offset F, an integer length L, a key buffer K corresponding to the current nugget, and an empty L-length XOR buffer.

SwitchCrypt expects the XOR buffer to be populated with L bytes of keystream output from some stream cipher seeked to offset F with respect to key K. The length of the key buffer will always be exactly what the cipher implementation expects, alleviating the burden of key management; similarly, the XOR buffer will be XOR-ed with the appropriate portion of nugget contents automatically, alleviating the burden of drive access and other tedious calculations.

2. **`read_interface`** and **`write_interface`**
Unlike `xor_interface`, encryption and decryption are distinct concerns at this abstraction level.
`read_interface` handles decryption during reads.
`write_interface` handles encryption during writes. Implementations receive full access to SwitchCrypt internals, giving wrapper code complete control over the encryption and decryption process and allowing implementers to bypass parts of the nugget-based drive layout abstraction (i.e. BODY) if necessary. This comes at the cost of 1) significantly increased code complexity, as the implementer must perform certain I/O manually, distinguish between independent nuggets on the drive, determine what to encrypt or decrypt at what offset and when, when to commit which metadata and where and 2) potential performance implications, since SwitchCrypt must account for not having absolute control over its internal data structures during function invocation. For a cipher like Freestyle, configurations with lower minimum and maximum rounds per block may see a performance improvement here, while configurations with higher minimum and maximum rounds per block may see reduced performance.

3.4. Cipher Switching Strategies

The Generic Stream Cipher Interface allows many differently ciphered nuggets to co-exist on the same drive. However, at any moment, there is only a single *active cipher configuration* (henceforth *active configuration*). The active configuration is used to encrypt nugget contents. When a cipher switch is triggered, a different configuration becomes the active configuration. At this point, SwitchCrypt must determine *when* to re-cipher a nugget and *where* to store the output on the drive. “Re-ciphering” here means using an inactive configuration to decrypt a nugget's contents and using the active configuration to re-cipher it. Depending on the use case, it may make the most sense to re-cipher a nugget immediately, or eventually, or to maintain several areas of differently-ciphered nuggets concurrently.

A naive approach would switch every nugget in BODY to the active configuration immediately, but the latency and energy cost would be unacceptable. Hence, a more strategic approach is necessary. We satisfy this need with our *cipher switching strategies*. These novel strategies allow for nuggets to be re-ciphered in a variety of cases with minimal impact on performance and battery life and without compromising security. This is thanks to the nugget-based drive layout, which limits the churn of cipher switching operations to relatively small regions of ciphertext on the drive.

Determining *when* to target a nugget for re-ciphering we call *temporal switching*, for which we propose the *Forward* switching strategy. Determining *where*—in which storage region and across which nuggets—to output ciphertext we call *spatial switching*, for which we propose the *Mirrored* and *Selective* switching strategies.

Forward Switching Strategy. When a nugget is encountered during I/O that was encrypted using something other than the active configuration, the Forward strategy dictates that this nugget be re-ciphered immediately. If a particular nugget encrypted with an inactive configuration is never encountered during I/O, it is never re-ciphered and remains on the drive in its original state. In this way, the Forward strategy represents a form of temporal cipher switching.

Rather than re-cipher the entire drive every time the active configuration changes, this strategy limits the performance impact of cipher switching to individual nuggets. The expense of re-ciphering is paid only once, after which the nugget is accessed normally during I/O until the active configuration is switched again.

Selective Switching Strategy. When SwitchCrypt is initialized with the Selective strategy, the drive is partitioned into C regions where C represents the total number of available ciphers in the system; each regions' nuggets are encrypted by each of the C ciphers respectively. For instance, were SwitchCrypt initialized using two ciphers ($C = 2$), the drive would be partitioned in half; all nuggets in the first region would be encrypted with the first cipher while all nuggets in the second would be encrypted with the other.

When using this strategy, the active cipher determines which partition we “select” for I/O operations. Hence, unlike the Forward strategy, which schedules individual nuggets to be re-ciphered at some point in time after the active configuration is switched, the Selective strategy allows the wider system to indicate *where* on the drive a read or write operation should occur. In this way, the Selective strategy represents a form of spatial cipher switching where different regions of the drive can store differently-ciphered nuggets independently and concurrently. A user could take advantage of this to, for instance, set up regions with different security properties and performance characteristics, managing them as distinct virtual drives or transparently reading/writing bytes to different security regions on the same drive.

Mirrored Switching Strategy. Similar to the Selective strategy, when SwitchCrypt is initialized with the Mirrored strategy, the drive is partitioned into C regions where C represents the total number of available ciphers in the system; each regions' nuggets are encrypted by each of the C ciphers respectively.

However, unlike the Selective strategy, all write operations that hit one region are mirrored into the other regions immediately, so all regions of the drive will always be in a consistent state and always share the same data. The active configuration determines *where* a read operation should occur. In this way, the Mirrored strategy represents a form of spatial cipher switching because we're switching which configuration we're using to read in data. A user could take advantage of this along with SSD Instant Secure Erase [1, 3, 28] to delete other regions, thus quickly and securely converging the drive to a single configuration without losing any data or suffering the egregious performance or battery penalty that comes with re-ciphering every nugget.

Strategy	Convergence	Waste	Performance
Forward	Slower	None	Faster reads and writes unless switching
Mirrored	Nearly instant	High	Faster reads; slower writes
Selective	Slower	High	Faster reads and writes

Table 2: A summary comparison between the three cipher switching strategies.

3.4.1. Comparing Cipher Switching Strategies Table 2 summarizes the higher level tradeoffs between the three cipher switching strategies.

Convergence. Depending on the use case, the ability to quickly converge the entire drive to a single cipher configuration without losing data is very useful (see: Section 5). The near-instantaneous "just forget the key" nature of SSD Instant Secure Erase (ISE) implementations on modern SSDs [1, 3, 28] makes this a very fast process for the Mirrored strategy. The Forward strategy is slow to converge compared to Mirrored since, in the worse case, every nugget on the drive will require re-ciphering. The Selective strategy is similarly slow to converge since entire regions of nuggets must be moved and re-ciphered to prevent data loss; those regions could be destroyed without moving data around using ISE too, which would be very fast, but unlike Mirrored some data would be lost forever.

Waste. Unlike the other two strategies, using the Forward strategy does not reduce the total usable space on the drive by the end-user, ciphertext expansion notwithstanding. We refer to this as "waste". The Forward strategy is not wasteful in this way because it allows differently-ciphered nuggets to co-exist contiguously on the drive without special partitions. Since the Mirrored and Selective strategies require partitioning the drive into some number of regions—where the writeable size reported back to the OS is some function of region size—there is a necessary reduction in usable space.

Performance. The Selective and Mirrored strategies can read data from the drive with low overhead, reaching performance parity with prior work, because they never have to deal with on-demand re-ciphering. This is because switching ciphers using these two strategies amounts to offsetting the read index so that it lands in the proper BODY partition on the drive, which has little overhead. The Forward strategy also reads with low overhead except in the case where a nugget was not encrypted with the active configuration. This triggers re-ciphering on-demand, which can be costly if the workload constantly touches unique nuggets and is small enough that cost is not amortized.

The Selective strategy also writes with low overhead because, like with reads, an index offset is the only requirement. The Mirrored strategy, on the other hand, can be up to two times slower for writes (when $C = 2$) compared to baseline. Each additional region ($C > 2$) compounds the write penalty depending on the workload. This is because each write is mirrored across *all* regions. As with reads, the Forward strategy writes with low overhead except in the case where a nugget was not encrypted with the active configuration. This triggers re-ciphering on-demand, which can be costly if the workload touches unique nuggets and is small enough that cost is not amortized.

With these tradeoffs in mind: Mirrored is ideal when the drive must converge quickly, write performance is not a primary concern, and drive space is abundant; Selective is ideal when different data should be encrypted differently and drive space is abundant; and Forward is ideal when some subset of nuggets should be encrypted differently without wasting drive space. See Section 5 for specific scenarios that demonstrate these differences in practice.

3.4.2. Threat Model for Cipher Switching Strategies The primary concern facing any FDE solution is that of confidentiality. An adversary should not be able to reveal any information about encrypted plaintext without the proper key. As with prior work, encryption is achieved via a binary additive approach: cipher output (keystream) is combined with plaintext nugget contents using XOR, with metadata to track writes and ensure that pad reuse never occurs during overwrites and that the system can recover from crashes into a secure state. Another concern is data integrity: an adversary should not be able to tamper with ciphertext and it go unnoticed. Nugget integrity is tracked by an in-memory Merkle tree. See the threat model addressed by Dickens et al. [17] for further details.

Switching strategies add an additional security concern not addressed by prior work: even if we initiate a "cipher switch," there may still be data on the drive that was encrypted with an inactive configuration. Is this a problem? For the Forward strategy, this implies data may at any time be encrypted using the "least desirable cipher". For the Mirrored and Selective strategies, the drive is partitioned into regions where nuggets are guaranteed to be encrypted with each cipher, including the "least desirable cipher". However, in terms of confidentiality, the confidentiality guarantee of SwitchCrypt can be reduced to the individual confidentiality guarantees of the available ciphers used to encrypt nuggets.

3.5. Putting It All Together

We revisit the motivating example from Section 2, where we're using Freestyle to ensure secure backups. Initially, I/O requests come down from the LFS and are received by the cryptographic driver, which divides the request based on which nuggets it touches. For each nugget, the per-nugget metadata is consulted to determine with which cipher the nugget is encrypted. If it is encrypted with the active cipher configuration (Freestyle), which must be true if we have not initiated a cipher switch, the write is handled similarly to prior work: encrypted data is read in from the drive, the merkle tree and monotonic counter are consulted to ensure the integrity of encrypted data, the transaction journal is consulted during write operations so that overwrites are handled and pad reuse violations are avoided, and then the keycount store is consulted to derive the nugget's unique encryption key from some master secret. Finally, using the Generic Stream Cipher Interface, we call out to the Freestyle, allowing SwitchCrypt to encrypts/decrypts the nugget's contents and commit any updates back to storage. All the while, the drive's Freestyle-encrypted contents are being uploaded up to our enterprise backup service every so often.

When the device enters "battery saver" mode, drive backups are paused, the energy monitoring software downclocks the CPU, and the OS signals to SwitchCrypt that a more energy-efficient cipher (ChaCha20) should be used until we return to a non-curtailed energy budget. SwitchCrypt sets ChaCha20 as the active cipher configuration. Now, when the cryptographic driver divides I/O requests into each affected nugget, the per-nugget metadata shows SwitchCrypt that each nugget is encrypted using a cipher that is not the active configuration. This triggers the re-ciphering code path. Since we are using the Forward switching strategy in this example, nugget data is immediately decrypted by calling out to the inactive configuration through the Generic Stream Cipher Interface, after which the nugget is re-ciphered by calling out to the active configuration. Finally, the cryptographic driver manages encrypting/decrypting data and updating the merkle tree and monotonic counter, transaction journal, and keycount store as the I/O operation and related metadata is committed to the drive afterwards.

Now, thanks to SwitchCrypt, the system can adapt to changing requirements beyond the capability of prior work. See Section 5 for specifics.

3.6. SwitchCrypt Implementation

Our SwitchCrypt implementation consists of 9,491 lines of C code; our test suite consists of 6,077 lines of C code. All together, our solution is comprised of 15,568 lines of C code and is publicly available open-source[1].

SwitchCrypt uses OpenSSL version 1.1.0h and LibSodium version 1.0.12 for its AES-XTS and AES-CTR implementations. Open source ARM NEON optimized implementations of ChaCha are provided by Floodyberry [18]. The Freestyle cipher reference implementation is from the original Freestyle paper [10]. The eSTREAM Profile 1 cipher implementations are from the open source libestream cryptographic library [25] by Lucas Clemente Vella. The Merkle Tree implementation is from the Secure Block Device [22].

We implement SwitchCrypt on top of the BUSE [8] virtual block device, using it as our mock device controller. BUSE is a thin (200 LoC) wrapper around the standard Linux Network Block Device (NBD). BUSE allows an operating system to transact block I/O requests to and from virtual block devices exposed via domain socket.

We develop Generic Cipher Interface wrapper implementations for many cipher implementations of which we select five for the purposes of this research. They are: ChaCha8 and ChaCha20 [12] as well as Freestyle [10] in three different configurations: a "fast" mode with parameters `FFast(`R_{min}=8, R_{max}=20, H_I=4, I_C=8`)`, a "balanced" mode with parameters `FBalanced(`R_{min}=12, R_{max}=28, H_I=2, I_C=10`)`, and a "strong" mode with parameters `FStrong(`R_{min}=20, R_{max}=36, H_I=1, I_C=12`)`.

4. Evaluation

4.1. Experimental Setup

We implement SwitchCrypt and our experiments on a Hardkernel Odroid XU3 ARM big.LITTLE system with Samsung Exynos 5422 A15/A7 heterogeneous multi-processing quad core CPUs at maximum clock speed, 2 gigabyte LPDDR3 RAM at 933 MHz, and an eMMC5.0 HS400 backing store running Ubuntu Trusty 14.04.6 LTS, kernel version 3.10.106. Energy monitoring was provided by the Odroid's integrated INA-231 power sensors polled every $\approx$ 260 milliseconds (not including noise/overhead).

We evaluate SwitchCrypt using a Linux RAM disk (tmpfs). The maximum theoretical memory bandwidth for this Odroid model is 14.9GB/s. Our observed maximum memory bandwidth is 4.5GB/s. Using a RAM disk focuses the evaluation on the performance differences due to different ciphers.

In each experiment below, we evaluate SwitchCrypt on two high level workloads: sequential and random I/O. In both workloads, a number of bytes are written and then read (either 4KB, 512KB, 5MB, 40MB) 10 times. Each workload is repeated three times for a total of 240 tests per cipher (720 runs per cipher pair/ratios, explained below); 30 results per byte size, 120 results per workload. Results are accumulated and the median is taken for each byte size.

When evaluating switching strategies, a finer breakdown in workloads is made using a pre-selected pair of ciphers we call the *primary cipher configuration* and *secondary cipher configuration*. SwitchCrypt is initialized at the primary configuration. Once we trigger a cipher switch, SwitchCrypt moves towards the secondary configuration via switching strategy.

The cipher switch is triggered according to a certain *ratio* of I/O operations. For example: given 10 40MB read-write operations, we may write and then read back 40MB 3 times using the primary cipher, initiate a cipher switch, and then write and then read back (write-read) 40MB 7 times. This would be a 3:7 ratio. It follows that there are three ratios we use to evaluate SwitchCrypt's performance in this regard: 7:3, 5:5, and 3:7. Respectively, that is 7 file write-read operation in the primary cipher for every 3 file write-read operations in the secondary cipher (7:3), 7 file write-read operation in the primary cipher for every other file write-read operation in the secondary cipher (5:5), and 3 file write-read operations in the primary cipher for every 7 file write-read operation in the secondary cipher (3:7).

All experiments are performed with basic Linux I/O commands, bypassing system caching.

In this section we answer the following questions:

1. What shape does the cipher configuration tradeoff space take under our workloads? (§4.2)
2. Can SwitchCrypt achieve dynamic security/energy tradeoffs by reaching configuration points not accessible with prior work? (§4.3)
3. What is the performance and storage overhead of each cipher switching strategy? (§4.4)

4.2. Switching Strategies Under Various Workloads

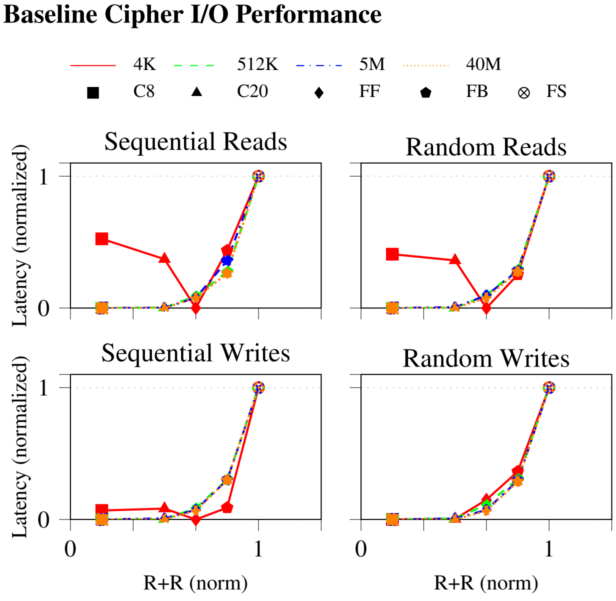

Figure 3: Median sequential and random write and then read performance baseline.

Fig. 3 shows the relative round count, ciphertext randomization, and ciphertext expansion scores versus median normalized latency tradeoff between different stream cipher configurations for our sequential and random I/O workloads. Trends for median hold when looking at tail latencies as well. Each line represents one workload: 4KB, 512KB, 5MB, and 40MB respectively (see legend). Each symbol represents one of our ciphers: ChaCha8, ChaCha20, Freestyle Fast, Freestyle Balanced, and Freestyle Strong. Of the ciphers we tested, those with higher round counts and higher ciphertext randomization scores resulted in higher latency and increased energy use for I/O operations. The relationship between these concerns is not simply linear, however, which exposes a rich tradeoff space.

Besides the 4KB workload, the shape of each workload follows a similar trend, hence we will focus on 40MB and 4KB workloads going forward. Due to the overhead of metadata management and the fast completion time of the 4KB workloads (*i.e.,* little time for amortization of overhead), ChaCha8 and ChaCha20 take longer to complete than Freestyle Fast. This advantage is not enough to make Freestyle Balanced or Secure workloads complete faster than the ChaCha variants, however.

Though ChaCha8 is faster than ChaCha20, there is some variability in our timing setup when capturing extremely fast events occurring close together in time. This is why ChaCha8 sometimes appears with higher latency than ChaCha20 for normalized 4KB workloads. ChaCha8 is not slower than ChaCha20.

4.3. Reaching Between Static Configuration Points

Fig. 4 shows the relative round count, ciphertext randomization, and ciphertext expansion scores versus median normalized latency tradeoff between different stream ciphers for our sequential and random I/O workloads with cipher switching using the Forward strategy. After a certain number of write-read I/O operations, a cipher switch is initiated and SwitchCrypt begins using the secondary cipher to encrypt and decrypt data. For each pair of ciphers, this is repeated three times: once at every ratio point *between* our static configuration points (*i.e.,* 7:3, 5:5, and 3:7 described above).

The point of this experiment is to determine if SwitchCrypt can effectively transition the drive between ciphers without devastating performance. For the 40MB, 5MB, and 512KB workloads (40MB is shown), we see that SwitchCrypt can achieve dynamic security/energy tradeoffs reaching points not accessible with prior work, all with minimal overhead.

Again, due to the overhead of metadata management for non-Freestyle ciphers (see §3.6) and the fast completion time of the 4KB workloads preventing SwitchCrypt from taking advantage of amortization, ChaCha8 and ChaCha20 take longer to complete than Freestyle Fast for 4KB reads. We also see very high latency for ratios between Freestyle Fast and Freestyle Strong cipher configurations. This is because Freestyle is not length-preserving, so extra write operations

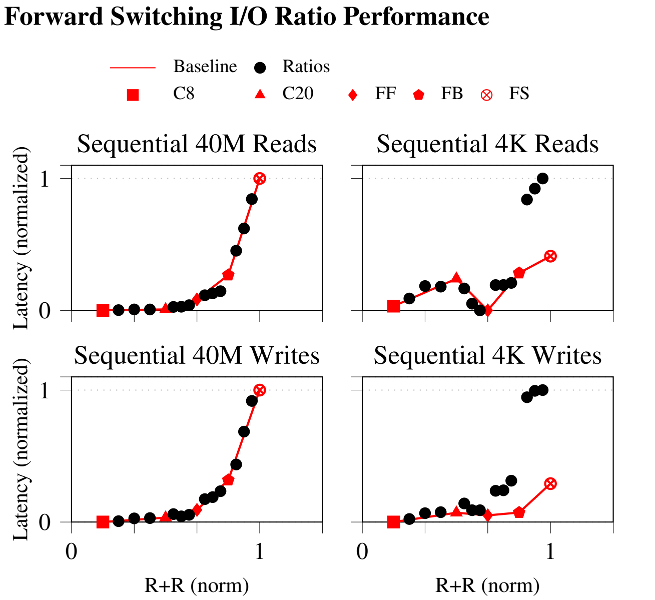

Figure 4: Median sequential and random write and then read back performance comparison of Forward switching to baseline. Each cluster of 3 dots between configurations represents the 7:3, 5:5, and 3:7 primary-vs-secondary I/O ratios described in the text.

must be performed, and the algorithm itself is generally much slower than the ChaCha variants (see Fig. 3). Doubly invoking Freestyle in a ratio configuration means these penalties are paid more often.

Fig. 5 show the performance of the Mirrored and Selective strategies with the same configuration of ratios as Fig. 4.

For the 40MB, 5MB, and 512KB workloads (40MB is shown), we see that Mirrored and Selective *read* workloads and the Selective *write* workload achieve parity with the Forward strategy experiments. This makes sense, as most of the overhead for Selective and Mirrored reads is determining which part of the drive to commit data to. The same applies to Selective writes. For the 4KB Mirrored and Selective *read* workloads and the Selective *write* workload, we see behavior similar to that in Fig. 4, as expected.

Mirrored writes across all workloads are very slow. This is to be expected, since the data is being mirrored across all areas of the drive. In our experiments, the drive can be considered partitioned in half. This overhead is most egregious for the 4KB Mirrored write workload. This makes Selective preferable to Mirrored. However, it's a tradeoff; Selective cannot quickly converge the drive to a single cipher configuration or survive the loss of an entire region.

4.4. Cipher Switching Overhead

We calculate that Forward switching has average overhead at 0.08x/0.10x for 40MB, 5MB and 512KB read/write workloads compared to baseline I/O, demonstrating SwitchCrypt's amortization of cipher switching costs. Average overhead is 0.38x/0.44x for 4KB read/write workloads when SwitchCrypt is unable to amortize cost. There is no spatial overhead with the Forward switching strategy.

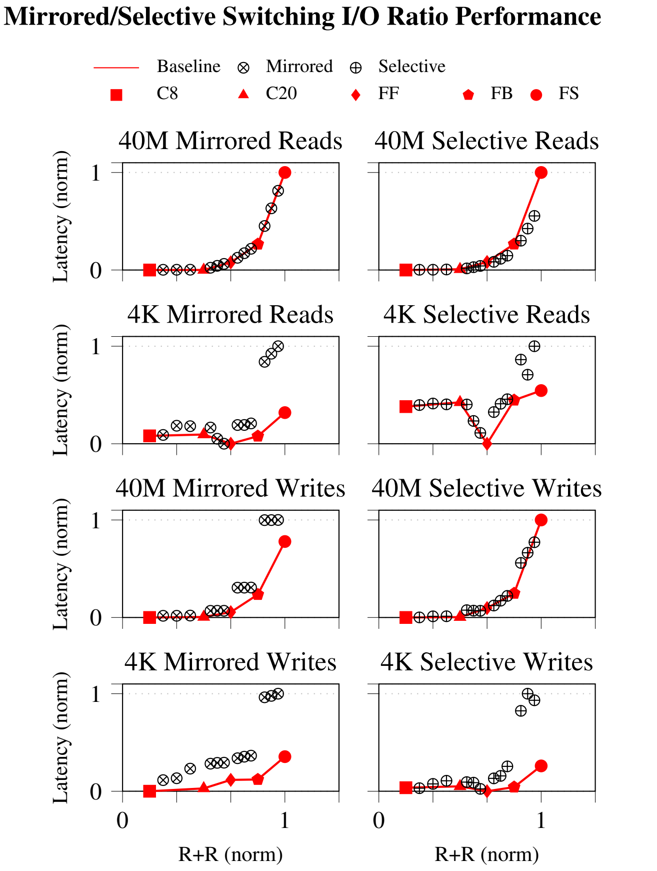

Figure 5: Median sequential and random write and then read back performance comparison of Mirrored and Selective switching strategies to baseline.

Similarly, we calculate that Selective switching has average overhead at 0x/0.3x for 40MB, 5MB and 512KB read/write workloads compared to baseline I/O. Average overhead is 0.22x/0.71x for 4KB read/write workloads. Spatial overhead in our experiment was half of all writable space on the drive.

Finally, we calculate that Mirrored switching has average overhead at 0.25x/0.61x for 40MB, 5MB and 512KB read/write workloads compared to baseline I/O, with high write latency due to mirroring. Average overhead is 0.55x/0.77x for 4KB read/write workloads. Spatial overhead in our experiment was half of all writable space on the drive.

These overhead numbers are the penalty paid for the additional flexibility of being able to reach configurations points that are unachievable without SwitchCrypt. SwitchCrypt's design keeps these overheads acceptably low in practice, achieving the desired goal of flexibly navigating latency/security tradeoffs for FDE.

5. SwitchCrypt Case Studies

In this section, we provide four case studies demonstrating the practical utility of cipher switching. We cover a wide range of situations, highlighting concerns like configuration convergence (§5.4), trading off security and writable space (§5.2), meeting latency goals (§5.3), and keeping within an energy budget (§5.1). We also demonstrate the utility of both temporal and spatial switching strategies, exploring the range of conditions under which certain strategies are optimal.

5.1. Balancing Security Goals with a Hard Energy Cap

Here we revisit the motivating example from Section 2. It illustrates that, because latency and energy use are correlated among the ciphers we examined, we can exploit that to save battery life when necessary while maintaining confidentiality when uploading encrypted backups to our enterprise solution.

To simulate I/O activity, we begin randomly writing 10 40MB files using the Freestyle Balanced cipher configuration over the course of approximately 120-125 seconds. After 5 seconds, the device enters "battery saver" mode, also pausing backup uploads. We simulate this event by 1) underclocking the cores to their lowest frequencies and 2) using `taskset` to transition the SwitchCrypt processes to the energy-efficient LITTLE cores. After this event, we complete the remaining workload using the ChaCha8 cipher. We repeat this experiment three times.

Battery Saver Use Case: Energy-Security Tradeoff vs Strict Energy Budget

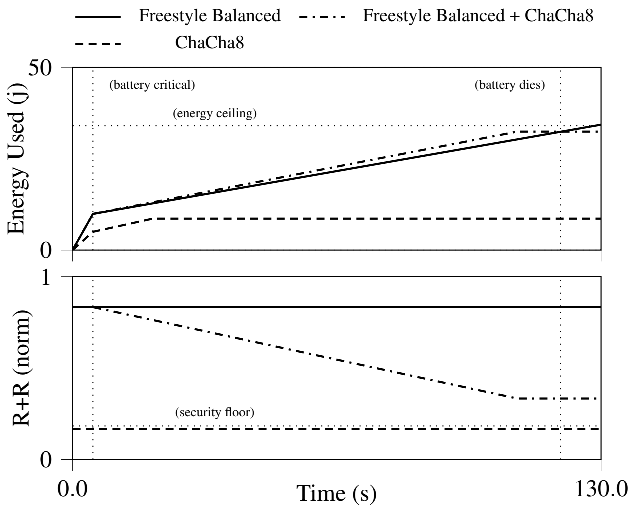

Figure 6: Median sequential write total energy use with respect to time with respect to time.

In Fig. 6, we see time versus energy used. At 0 seconds, we begin writing. At 5 seconds, the "battery saver" event occurs, causing the system to be underclocked. At 120 seconds, the system will die. If we blow past our energy ceiling, the system will die.

Our goal is to complete I/O before the device dies. We have three cipher configuration choices. 1) Favor security even when backups are paused and use Freestyle Balanced exclusively. Our results show that the device will die before I/O completes. 2) Favor low energy use even when backups are being uploaded and use ChaCha8 exclusively. Our results show that the device will finish writing early, but we cannot safely upload backups when nuggets are encrypted using ChaCha8. Finally, we have 3) favor security and use Freestyle Balanced when uploading backups, and switch to ChaCha8 when the system enters the low power state. Our results show that, while the system uses slightly more power in the short term, we stay within our energy budget and finish before the devices dies. Further, when we get our device to a charger, SwitchCrypt can converge nuggets back to Freestyle Balanced and resume uploading backups.

On average, using Forward cipher switching resulted in a 3.3x total energy use reduction compared to exclusively using Freestyle Balanced, allowing us to remain within our energy budget. We note, however, that the energy savings is not the point of this experiment. Rather, the lesson learned is that SwitchCrypt enables the system to move to the right point in the energy/security tradeoff space so that the current task can still be accomplished before the battery is drained and without compromising backup security at any point.

5.2. Variable Security Regions

This usecase illustrates utility of spatial Selective switching to achieve a performance win over prior work where the entire drive is encrypted with a single cipher. We demonstrate *Variable Security Regions* (VSR), where we can choose to encrypt select files or portions of files with different keys and ciphers below the filesystem level.

Storing classified materials, corporate secrets, etc. require the highest level of discretion, yet sensitive information like this can appear within a (much) larger amount of data that we value less. But if only a small percentage of the data needs the strongest encryption, then only a small percentage of the data should have that associated overhead. In this scenario, a user wants to indicate one or more regions of a file are more sensitive than the others. For example, perhaps banking transaction information is littered throughout a PDF; perhaps passwords and other sensitive information exists within several much larger files. Using prior techniques, either all the data would be stored with high overhead, the critical data would be stored without the mandated cipher type, or the data would have to be split among separate files requiring potentially complex and error-prone management schemes. SwitchCrypt VSRs allows us to sidestep these issues.

We begin by with 10 5MB and 4KB write-read operations to two SwitchCrypt instances: one using ChaCha8 (C8) and the other using Freestyle Strong (FS). These results represent I/O without cipher switching where 100% of the data is stored with either high overhead (FS) or using an inappropriate cipher

(C8). We then use a third SwitchCrypt instance initialized with Selective switching and write-read with a 7:3 ratio of ChaCha8 versus Freestyle Balanced I/O operations. Here, 30% of the data is considered sensitive. We repeat this experiment three times.

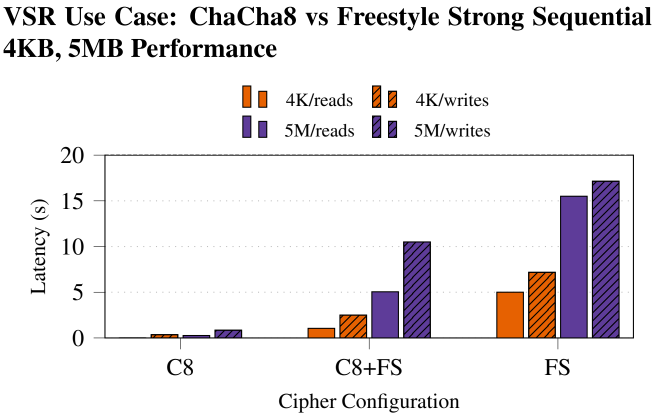

Figure 7: Median sequential read and write performance comparison of 4KB, 5MB I/O with 7-to-3 ratio, ChaCha8 vs Freestyle Strong (respectively).

In Fig. 7, we see the sequential read and write performance of 4KB and 5MB workloads when nuggets are encrypted exclusively with ChaCha8 or Freestyle Balanced. Between them, we see Selective switching 7:3 ratio I/O results.

Our goal is to use VSRs to keep our sensitive data at the mandated security level while keeping the performance and battery life benefits of using a fast cipher for the majority of I/O operations. Using SwitchCrypt Selective switching versus prior work results in a reduction of 3.1x to 4.8x for read latency and 1.6x to 2.8x for write latency, all without compromising the security needs of the most sensitive data.

5.3. Responding to End-of-Life Slowdown in SSDs

This usecase illustrates using temporal Forward switching to offset the debilitating decline in performance when SSDs reach end-of-life (EoL) [21]. We demonstrate the utility of such a system to dynamically stay within a strict latency budget while meeting minimum security requirements, which is not possible using prior work.

Due to garbage collection and wear-leveling requirements of SSDs, as free space becomes constrained, I/O performance drops precipitously [21]. With prior work, our strict latency ceiling is violated. However, if SwitchCrypt is made aware when the drive is in such a state, we can offset some of the performance loss by switching the ciphers of high traffic nuggets to the fastest acceptable cipher available using Forward switching.

We begin by writing 10 40MB files to SwitchCrypt per each cipher as a baseline. We then introduce a delay into SwitchCrypt I/O of 20*ms*, simulating drive slowdown, and repeat the experiment three times.

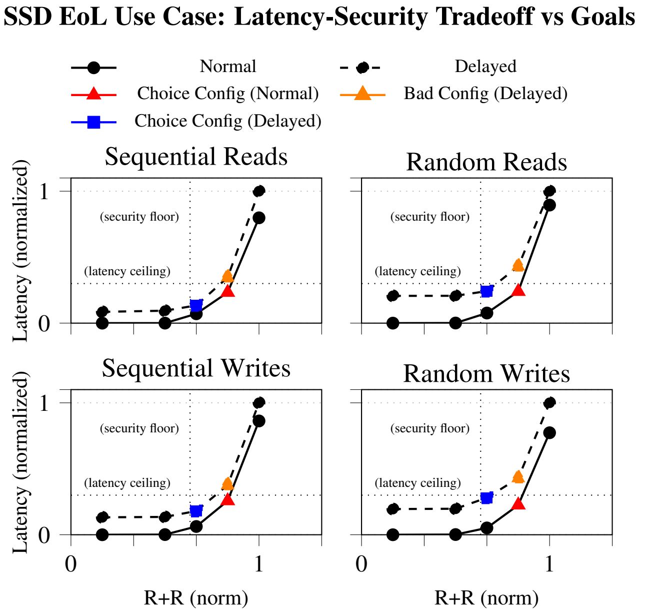

Figure 8: Median sequential and random 40MB read and write performance comparison: baseline versus simulated faulty block device.

In Fig. 8, we see the sequential and random read and write performance of a 40MB workload when nuggets are encrypted exclusively with our choice ciphers. While the latency ceiling and security floor have not changed, we see increased latency in the delayed workloads.

Our goal is to remain under the latency ceiling while remaining above the security floor. Thanks to Forward switching, accesses to highly trafficked areas of the drive can remain performant even during drive end-of-life.

5.4. Custody Panic: Device Data Under Duress

This usecase illustrates the utility of spatial Mirrored switching to take advantage of more energy-efficient high-performance ciphers while retaining the ability to quickly converge the entire drive to a single high-security cipher leveraging SSD Instant Secure Erase (ISE).

Nation-state and other adversaries have extensive compute resources, knowledge of obscure side-channels and back doors (*e.g.,* Dual_EC_DRBG [35]), and access to technology like quantum computers. Suppose a scientist were attempting to re-enter her country through a border entry point when she is stopped. Further suppose her laptop containing sensitive priceless research data (replicated across multiple SSDs via the SwitchCrypt Mirrored strategy) is confiscated from her custody. Being a security researcher, she has a chance to trigger a "remote wipe," where the laptop's SSD uses Instant Secure Erase (ISE) to "forget" the key material used by the black-box FTL/drive controller's anti-wear-leveling measures, supposedly rendering all her drives' data inaccessible [1, 3, 28].

However, while our scientist certainly does not want her data falling into the wrong hands, she cannot afford to permanently lose access to that data either, which triggering storage-wide ISE would accomplish. In such a scenario, it would be useful if she could trigger ISE on all of the mirrored drives except the one with the strongest cipher configuration. Our goal, then, is to slow down any attacker as much as possible such that, even if they permanently store her encrypted data off-site for later attempts at decryption [?] with exponentially more powerful compute resources and other new technologies, that data is more likely to remain irrecoverable. This allows her to use her device in her daily life without suffering the egregious performance penalties of exclusively using the strongest cipher configuration while also allowing the system to instantly converge to the strongest possible state in an emergency.

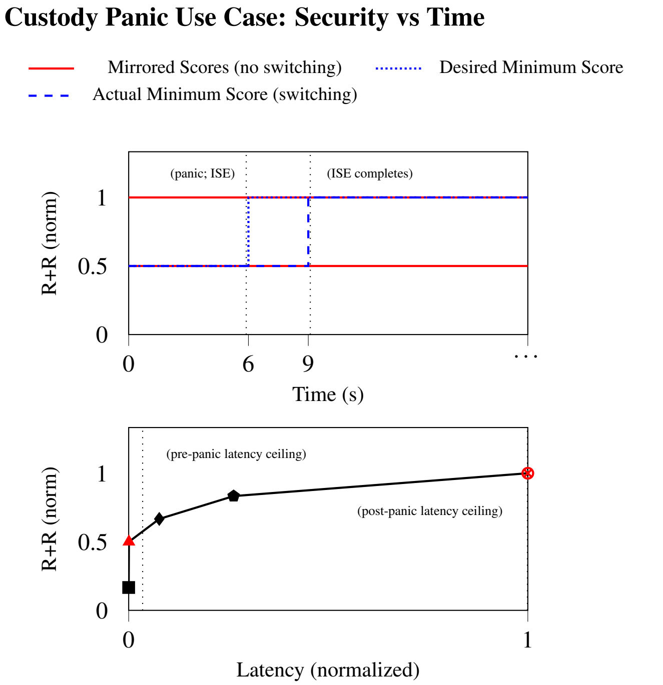

Figure 9: Actual security state vs desired security state with respect to the time and ISE.

In Fig. 8, we see the system begins at 0 seconds, where all data is mirrored across partitions. Both the desired and minimum security state of the drive is 1.5, a balance between performance and security. For simplicity, we define "security state" here as the simple sum of two scores: *relative round count and ciphertext randomization*. At 6 seconds, custody panic is triggered—the desired minimum security state goes to 3, the highest possible—at which point the system executes ISE and completely erases the minimally encrypted partition. ISE, which takes advantage of FTL/drive controller level encryption to is known to be much faster than TRIM and completes in as little as 3 seconds [1, 3, 28]. Once complete, the most secure form of the data is all that remains. The backing store has been “locked down.”

We show that, given a device that supports SSD ISE, SwitchCrypt, and the Mirrored strategy, we can quickly and practically converge the backing store to this locked down state. With prior work, data is either encrypted inappropriately or the device becomes too slow for daily use (latency ceiling). In exchange, we trade off half of our drive’s writeable space.

6. Related Work

The standard approach to FDE, using AES-XTS, introduces significant overhead. Recently, it has been established that encryption using *stream ciphers* for FDE is faster than using AES [17], but accomplishing this in practice it is non-trivial. Prior work explores several approaches: a non-deterministic CTR mode (Freestyle [10]), a length-preserving “tweakable super-pseudorandom permutation” (Adiantum [15]), and a stream cipher in a binary additive (XOR) mode leveraging LFS overwrite-averse behavior to prevent overwrites (StrongBox [17]).

Unlike StrongBox and other work, which focuses on optimizing performance despite re-ciphering due to overwrites, SwitchCrypt maintains overwrite protections while abstracting the idea of re-ciphering nuggets out into cipher switching; instead of myopically pursuing a performance win, we can pursue energy/battery and security wins as well.

Further, trading off security for energy, performance, and other concerns is not a new research area [19, 32, 33, 20, 24, 27, 34]. Goodman et al. introduced selectively decreasing the security of some data to save energy [19]. However, their approach is designed for communication and only considered iteration/round count, thus it did not anticipate the need for SwitchCrypt’s generic interface, switching strategies, or quantification framework. Wolter and Reinecke study approaches to quantifying security in several contexts [32]. This study anticipates the value of dynamically switching ciphers but proposes no mechanisms to enable this in FDE. Similarly, companies like LastPass and Google have explored performance-security tradeoffs. Google’s Adiantum uses a reduced-round version of ChaCha in exchange for performance [15]. While not an FDE solution, LastPass has dealt with scaling the number of iterations of PBKDF#2, trading performance for security during login sessions [5].

7. Conclusion

This paper advocates for a more flexible approach to FDE where the storage system can dynamically adjust the tradeoffs between security and latency/energy. To support this vision of agile encryption, we proposed an interface that allows multiple stream ciphers with different input and output characteristics to be composed in a generic manner. We have identified three strategies for using this interface to switch ciphers dynamically and with low overhead. We have also proposed a quantification framework for determining when to use one cipher over another. Our case studies show how different strategies can be used to optimize for different goals in practice. We believe that agile encryption will become increasingly important as successful systems are increasingly required to balance conflicting operational requirements. We hope that this work inspires further research in achieving this balance. Our work is publicly available open-source[1].

References

[1]

[2] Android open source project: Full-disk encryption.

[3] Seagate instant secure erase deployment options.

[4] Using advanced encryption standard counter mode (aes-ctr) with the internet key exchange version 02 (ikev2) protocol.

[5] What makes lastpass secure?

[6] The xts-aes tweakable block cipher, 2008. IEEE Std 1619-2007.

[7] Recommendation for block cipher modes of operation: The xts-aes mode for confidentiality on storage devices, 2010. NIST Special Publication 800-38E.

[8] A block device in userspace, 2012.

[9] Linux kernel device-mapper crypto target, 2013.

[10] P. Arun Babu and Jithin Jose Thomas. Freestyle, a randomized version of chacha for resisting offline brute-force and dictionary attacks. Cryptology ePrint Archive, Report 2018/1127, 2018. https://eprint.iacr.org/2018/1127.

[11] Côme Berbain, Olivier Billet, Anne Canteaut, Nicolas Courtois, Henri Gilbert, Louis Goubin, Aline Gouget, Louis Granboulan, Cédric Lauradoux, Marine Minier, Thomas Pornin, and Hervé Sibert. *Sosemanuk, a Fast Software-Oriented Stream Cipher*, pages 98–118. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[12] Daniel J. Bernstein. Chacha, a variant of salsa20. Technical report, University of Illinois at Chicago, 2008.

[13] Daniel J. Bernstein. The salsa20 family of stream ciphers. In *The eSTREAM Finalists*, 2008.

[14] Martin Boesgaard, Mette Vesterager, and Erik Zenner. *The Rabbit Stream Cipher*, pages 69–83. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[15] Paul Crowley and Eric Biggers. Adiantum: length-preserving encryption for entry-level processors. *IACR Transactions on Symmetric Cryptology*, 2018(4):39–61, 2018.

[16] Andrew Cunningham and Utc. Google quietly backs away from encrypting new lollipop devices by default, Mar 2015.

[17] Bernard Dickens III, Haryadi S. Gunawi, Ariel J. Feldman, and Henry Hoffmann. Strongbox: Confidentiality, integrity, and performance using stream ciphers for full drive encryption. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '18, pages 708–721, New York, NY, USA, 2018. ACM.

[18] Floodyberry. floodyberry/chacha-opt, Mar 2015.

[19] James Goodman, Abram P Dancy, and Anantha P Chandrakasan. An energy/security scalable encryption processor using an embedded variable voltage dc/dc converter. *IEEE Journal of Solid-State Circuits*, 33(11):1799–1809, 1998.

[20] M. Haleem, C. Mathur, R. Chandramouli, and K. Subbalakshmi. Opportunistic encryption: A trade-off between security and throughput in wireless networks. *IEEE Transactions on Dependable and Secure Computing*, 4(4):313–324, Oct 2007.

[21] Mingzhe Hao, Gokul Soundararajan, Deepak Kenchammana-Hosekote, Andrew A Chien, and Haryadi S Gunawi. The tail at store: A revelation from millions of hours of disk and ssd deployments. In *14th {USENIX} Conference on File and Storage Technologies ({FAST} 16)*, pages 263–276, 2016.

[22] 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*, volume 1, pages 222–229, 2015.

[23] Joshua Ho and Brandon Chester. Encryption and storage performance in android 5.0 lollipop, Nov 2014.

[24] Jun Li and Edward R. Omiecinski. Efficiency and security trade-off in supporting range queries on encrypted databases. In Sushil Jajodia and Duminda Wijesekera, editors, *Data and Applications Security XIX*, pages 69–83, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[25] Lvella. lvella/libestream.

[26] Subhamoy Maitra. Chosen iv cryptanalysis on reduced round chacha and salsa. Technical report, Applied Statistics Unit, Indian Statistical Institute, 2015.

[27] Andreas Merkel and Frank Bellosa. Balancing power consumption in multiprocessor systems. In *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006*, EuroSys '06, pages 403–414, New York, NY, USA, 2006. ACM.

[28] Tilo Müller, Tobias Latzo, and Felix C Freiling. Self-encrypting disks pose self-decrypting risks. In *the 29th Chaos Communinication Congress*, pages 1–10, 2012.

[29] NIST. Public comments on the xts-aes mode, 2008.

[30] Phillip Rogaway. Efficient instantiations of tweakable blockciphers and refinements to modes ocb and pmac. In Pil Joong Lee, editor, *Advances in Cryptology - ASIACRYPT 2004*, pages 16–31, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[31] Timothy J. Seppala. Google won't force android encryption by default, Jul 2019.

[32] Katinka Wolter and Philipp Reinecke. Performance and security tradeoff. In *International School on Formal Methods for the Design of Computer, Communication and Software Systems*, pages 135–167. Springer, 2010.

[33] W. Zeng and M. Chow. Optimal tradeoff between performance and security in networked control systems based on coevolutionary algorithms. *IEEE Transactions on Industrial Electronics*, 59(7):3016–3025, July 2012.

[34] W. Zeng and M. Chow. Modeling and optimizing the performance-security tradeoff on d-ncs using the coevolutionary paradigm. *IEEE Transactions on Industrial Informatics*, 9(1):394–402, Feb 2013.

[35] Kim Zetter. How a crypto 'backdoor' pitted the tech world against the nsa, Jun 2017.