FLEXCRYPT: Kernel Support for Heterogeneous Full Drive Encryption

Paper # ??

Abstract

[TODO: this section]

1 Introduction

Security is a very important property of storage systems. Decades of systems and storage research in this space have looked into security in the context of V [**?**, **?**, **?**], W [**?**, **?**, **?**], X [**?**, **?**, **?**], Y [**?**, **?**, **?**], Z [**?**, **?**, **?**], and many more.

For local storage, the state of the art for securing data at rest, such as the contents of a laptop's SSD, is Full Drive Encryption (FDE). Unfortunately, encryption introduces overhead that drastically impacts system performance and energy consumption. FDE implementations such as dm-crypt [2, 8] for Linux and BitLocker for Windows [**?**, **?**] are considered the standard available solutions. Behind these implementations is the slow AES *block cipher* (AES-XTS) [5, 6, 22]. In this regard, the story of 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 [15]. 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" [24]. 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 [18].

Fortunately, in the last two years, there have been major advancements in the FDE technology that distances away from the slow block cipher technology and successfully implements *stream* chipers such as ChaCha20 to full device encryption. Recent works—such as Google's HBSH (hash, block cipher, stream cipher, hash) [**?**], Adiantum [14], and StrongBox [16]—brings stream cipher based FDE to devices that do not or cannot support hardware accelerated AES. A key to efficiency adopt stream chiper into the storage layer is to pair it with a log-structured file system such as F2FS on flash devices (because **(HSG$_0$: explain in one sentence)**).

These advancements open up a new opportunity that: *can a file/storage system support multiple or flexible switching of full drive ecnryption?* We postulate that such a feature is really needed but today we find *no* operating systems that can support such a feature. To motivate this feature, let's think about this one scenario: local files in a user's mobile phone are "forced" to use encryption CRYPTA because it is one of the highest secured encryption that is safe for backing up the local files to the cloud. However when the mobile phone runs out of battery and is doing a lot of I/Os, the heavy encryption will drain the battery down. The user might wish a low battery mode where the data being accessed is convered into a less powerful encryption momentarily.

We present FLEXCRYPT, to the best of our knowledge, the *first* kernel support (at the block level) that provides file systems with flexible switching of full drive encryption. As FDE's impact on drive performance and energy efficiency depends on a multitude of choices, different ciphers expose different performance and energy efficiency characteristics. FLEXCRYPT allows cipher choice to be be viewed as a key configuration parameter, as opposed to a static choice at format or boot time. FLEXCRYPT allows "encryption" to adapt to changes that arise while the system is running, including changes in resource availability, runtime environment, desired security properties, and respecting changing OS energy budgets. FLEXCRYPT allows users to perform cipher switching in space and time (*e.g.,* , more secure files and temporal switching). FLEXCRYPT allows the software system to navigate the tradeoff space made by balancing competing security and latency requirements. **(HSG$_1$: This is where we list all the benefits; try not to be redundant)** To achieve all these benefits, FLEXCRYPT comes with three important elements, representing the three main contributions of the paper.

First, we introduce FLEXSWITCH, a kernel configuration that exposes three types of switching models *forward*, *selective* and *mirrored*, to to "re-cipher" storage units dynamically, allowing us to tradeoff different performance and security properties of various configurations at runtime. These switching models define what the I/O layer should do upon the ongoing read/write I/Os during the switching. These three switching models are motivated from real case studies. For examples, forward switching is motivated from the battery case study above; selective switching is motivated from cases where users desire to have certain files (*e.g.,* , legal documents) much more secure than the others; and mirrored switching is motivated for server-side cases that would like to perform "encryption upgrade" without zero downtime.

Second, to supports the switching models above, we implement FLEXCRYPTS, a block-level module that contains encryption implementations ("crypts") that have been restructured to support switching. Prior works mainly implement one cipher choice and the implementation is very much integrated

with the file/block layer [**?**, 16]. The key challenge to support multiple ciphers is that different chipers take different inputs and produce different type of outputs. For examples, CRYPTA requires nonce input, CRYPTB config(??**xxx**) and CRYPTC sector info and CRYPTC outputs streams while the others output **xxx** . Thus, we introduce a novel design substantively expanding prior FDE work by wholly decoupling cipher implementations from the encryption process. In FLEXCRYPTS, we wrote hooks that manages the required input values and the output format of different ciphers.

Finally, we initiate FLEXTRADE, a scheme that attempts to *quantify* the tradeoffs in the the rich configuration space of stream ciphers. 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. FLEXTRADE helps users in understanding their cipher choices as they come with a variety of performance, energy efficiency, and security properties in the FDE context.

We performed a comprehensive evaluation in several ways. First, we show that FLEXCRYPT successfully supports a wide variety of ciphers; specifically we have integrated *7 ciphers* and a total of *15 cipher configurations* into FLEXCRYPT in **xxx** as a kernel block module (will be open-sourced), and can act as an off-the-shelf replacement for dm-crypt. The ciphers are ChaCha8 and ChaCha12 [11], Freestyle [9]), SalsaX [12], AES in counter mode (AES-CTR) [4], Rabbit [13], Sosemanuk [10], **xxx** . Second, we showcase the benefits of FLEXCRYPT with experiments illustrating three real-world case studies (battery low case, file-level protections, and no downtime) and show the performance/energy tradeoffs. Finally, we perform several benchmarking to show the performance and **xxx** of the individual ciphers and the switching overhead of the three switching models we provide **(HSG$_2$: ???)**.

2 Background and Motivation

We provide some case studies that motivate our work and background materials about implementation of stream cipher on log-structured file systems.

File encryption reacting to battery saver

2.1 The Static FDE Problem

The major issue with full drive encryption (FDE) is its static nature; users must commit one FDE configuration without much flexibility. Let us suppose we have an ARM-based ultra-low-voltage netbook provided to us by our employer who requires that the drive is fully encrypted at all times and is constantly backed up to an offsite system. Given that the FDE industry standard is AES-XTS, we initialize our drive with it. Here, three primary concerns present themselves: performance, vulnerability and inflexibility.

First, it is well known that AES-XTS adds significant latency and power overhead to I/O operations, especially on mobile and battery-constrained devices [15, 18, 24]. As this scenario requires the drive encrypted at all times, we must accept this hit to performance and battery life. Worse, if our device does not support hardware accelerated AES (which is hardly ubiquitous) performance can be degraded even further; I/O latency can be as high as 3–5x [16].

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. However, access to multiple snapshots of a drive's AES-XTS-encrypted contents presents a vulnerability—an attacker can passively glean information about the plaintext over time by contrasting those snapshots, leading to confidentiality violations in some situations [5, 23]. In this case, we might want to employ a strong encryption solution such as stream ciphers, but they are known to perform worst in many file systems.

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.

2.2 Recent Advancements

In this paper we categorize two types of FDE: *block* and *stream* chipers. Block chipers were a defacto solution for storage while stream chipers were prevalent in networking **(HSG$_3$: true??)**. Their main differences are **(HSG$_4$: fill here, 1 or 2 sentences)**. There have been major advancements in the FDE technology that distances away from the slow block cipher technology and successfully implements stream chipers full device encryption [**?**, 14, 16] even without hardware accelerated encryption.

There are two technological shifts that enable this confidential, high-performance storage with stream ciphers. First, devices today commonly employ solid-state storage with Flash Translation Layers (FTL), which operate similarly to Log-structured File Systems (LFS) [**?**, **?**, **?**]. The no inplace update nature of FTL or LFS-like storage allows the overwrite-heavy stream chipers to be efficiently implemented (*e.g.,* , ChaCha with F2FS or LogFS performs **xxx** times better than with ext4 [16]). Second, mobile devices now support trusted hardware, such as Trusted Execution Environments (TEE) [**?**, **?**] and secure storage areas [**?**] which means the drive encryption module has access to persistent, monotonically increasing counters that can be used to prevent rollback attacks when overwrites do occur.

2.3 No One-Size-Fits-All Case Studies

As the recent advancements show that many different ciphers can be implemented in efficient manner, this opens up an opportunity for the storage layer to support a wide variety of encryption technologies. Clearly, many case studies show that there is no one-size-fits-all encryption, and some flexibility is demanded. Below we present three real-wrold case studies.

Battery-life saver. In the example above, in certain situations, the user might want to prioritize reducing the total energy use. For example, while in battery low mode, the kernel now switches to a more energy-efficient encryption and pauses the offline cloud backup moementarily. Users might be willing to accept this tradeoff and violate the "backup at all time" rule, knowing that within a few hours the user will have access to power (no difference from internet connection problem). This case study is pervasive [**?**, **?**].

Cipher upgrade without downtime. From mobile devices, we now turn to server-side storage that often require encryption upgrade [**?**]. A server provider might decide to completely upgrade from a encryption technology that has been broken into a stronger encryption technology. Ideally during the switch, the server still can continuously serves users without any downtime. However, without kernel-level support, the server provider must write an application level software that performs the whole switching operation and manually redirects users to the appropriate files.

Select files. Learning from the wireless literature, it is advocated that certain files that traverse the Internet or wireless connection in particular, should be encryption with a more secure way. Example of those files include government documents [**?**], credit card information [**?**], xxx [**?**, **?**, **?**].

3 Overview

4 SwitchCrypt Design

FLEXCRYPT is a block layer kernel module that can replace other state-of-the-art block-level encryption technologies such as the popular dm-crypt. FLEXCRYPT does not require any modification in the application and only a small modification in the file system layer to expose the inode-block mapping. The choice to do this at the block layer is importnat to ensure the core logic of file systems does not have to be modified. Just like any other encryption tehcnologies, FLEXCRYPT must have a unit of en/decryption. In this paper we call it a "nugget," which can be configured as one or more sequential blocks.

To provide a kernel support for different cipher configurationsa and live cipher switching, we must address three key challenges:

1. We must provide switching models, both that cover temporal and spatial nature of storage data and accesses, to support live switching that users demand. For this, we introduce the FLEXSWITCH (Section **??**) component of FLEXCRYPT that exposes three switching models to users: forward, selective and mirrored switching.
2. We must allow different cipher implementations to be easily integrated to FLEXCRYPT, however different ciphers require different types of inputs and outputs and often they are tighthly integrated to the encrypted data and the on-disk data structures that the encryption layer uses. For this, we introduce the FLEXCRYPTS (Section **??**) component where we decouple cipher implementations from the core encryption algorithm and provide interfaces that allows many different encryption algorithms ("crypts") to be rewritten easily.
3. We must help users understand the tradeoffs of different cipher configurations, hence we introduce FLEXTRADE (Section **??**), a method that attempts to quantify the desirable properties of different cipher configurations in several metrics such as the round-trip, randomization and expansion costs.

4.1 FLEXSWITCH: Cipher Switching Models

Just like typical storage devices, at any moment, there is only *one active* cipher configuration (henceforth "active configuration"). However, with FLEXSWITCH, we provide a small temporal and spatial flexibility that allows movement from one configuration point to another or even settle on optimal configurations wholly unachievable with prior work. When a cipher switch is triggered, a different configuration becomes the active configuration, and "rechipering" must be done — using the just-inactived configuration to decrypt a nugget's contents and using the active configuration to re-cipher it.

The challenge here is to accomplish this while minimizing overhead. A naive approach would switch every nugget in the device to the active configuration immediately, but the latency and energy cost would be unacceptable. Hence, a more strategic approach is to provide different switching models that allow higher-level policies to choose; for example, 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. The different models allow for nuggets to be re-ciphered in a variety of cases with minimal impact on performance and battery life and without compromising security. We introduce three switching strategies: temporal, mirrored, and selective. The first one forms a temporal switching while the latter two form a spatial switching, as explained below and summarized in Table **??**.

(HSG_5: If I explain things wrong here, go ahead fix it. This is based on my best understanding of our conversation. Also, discuss with Hank if we should just name "Forward" with "Temporal".)

Forward switching. The first type of cipher switching we support is for the battery-life example where we want to switch from cipher "C_1", a highly-secure, energy-expensive cipher to cipher "C_2," less-secure but more energy-efficient

```
// Change this with a real table, use paragraph
// style for cell content (I assume you know how)
           Read                    Write
-----------------------------------------------------
Forward    Not-ciphered            Ciphered with B,
           if 1st read             Reciphered on demand
           Reciphered with B on    in the future when
           2nd read                A is active again
-----------------------------------------------------
Mirrored   Read from the prev      Duplicated both
           cipher's region if      in the prev and
           during migration.       new regions until
           Read from the new       migration completes
           region if migration
           completes
-----------------------------------------------------
```

Table 1: **Reciphering/switching modes.** *The table explains for every mode what happens on I/Os when the switch happens from cipher A to cipher B. "Read" means read of existing data during the switch. "Write" means new.*

one. This mode can be enabled with a battery-saving mode (the higher-level policy) supported by the OS.

Table **??** summarizes what happens during the switch. For new incoming writes, the data will be ciphered with C_2. Later on, when the temporal switching ends, the data will remain ciphered with C_2**(HSG$_6$: correct?)**. The reason we don't recipher it back to C_1 in the future is that the files might be just a temporary file (*e.g.,* , movie download) that is only read once. However, if the data is read again in the future when C_1 is active again (*e.g.,* , for backing up to the cloud), the data will then reciphered to C_1. For read operations, if the existing data (ciphered with C_1) is read during the switch, the data will be reciphered to C_2. The reason behind this is that reciphering to C_2 is not expensive, and better to be done on demand on the first read (during the switching) than later. In general, forward switching limits the performance impact of cipher switching to individual nuggets being accessed. The data at rest (ciphered with C_1) that is never accessed during the switch remains in its original state.

We note that there can be variants to the forward switching mode. The above concept favors performance. Another variant that can be support that favors stronger security can be done this way: For writes, the moment the switch back from C_2 to C_1 happens, the new data that was ciphered with C_2 is quickly ciphered to C_1. To do this, we need to do more recording, while the one we proposed above can be done on demand (when the data is read in the future). Another variant for the read operations can also be done as follows: the data will *not* be reciphered on the first read. The intuition is that forward switching only happens temporarily (while the battery is low), hence data being read might only be read once and stay in the memory. However, if the data is read the second time (during the temporal period), the data is reciphered to C_2. These and other possible variants can be left for future work. So far, we find our version of forward switching suits a common battery-life scenario.

Mirrored Switching. Next, we consider a different scenario where "C_1" is a recently-attacked and vulnerable encryption and "C_2" is a newly recommended cipher that's just been added to FLEXCRYPT's latest kernel/module upgrade. Let us imagine a server-side storage operator who wants to switch from C_1 to C_2 without any server downtime. To achieve this, FLEXCRYPT since the beginning will partition the drive into 2 regions. That is, to support a full-drive cipher switch (supportec by the block layer and without application/file system modification), we must pay the space cost to anticipate such a switch in the future. In this mode, we must add the "migration" period to reflect the transition window from C_1 to C_2.

(HSG$_7$: I kind of change a bit how this mode should perform. To me the previous method – that partitions the device with C regions, it's a showstopper. Imagine if you support C=10, are you saying that now we lose 90% of the space for user data, and we keep them all mirroring(??). That's a no no. Also the case study here is about switching to a better cipher; so you cannot anticipate way before what kind of new cipher you'll transition into. Also, talk to Hank if another name like "Full/Migration/??" Switching is a better one– because we're only mirroring during the migration.)

During the migration, as summarized in Table **??**, all write operations that hit the C_1's region will be mirrored to the C_2's region. At the same time, the data in C_1's region is reciphered incrementally to C_2's region. Still during the migration, read operations will be served by the original state (C_1's region) until the full switch happens. This is essentially similar to VM migration [**?**]. After the whole migration completes, to fully secure-erase the previous region, one can use a mechanism such as SSD Instant Secure Erase [1, 3, 21], 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.

We would like to note that it is fundemantally hard to have two cipher configurations enabled on the same drive. There are two reasons. First, ..**(HSG$_8$: anything about data structures, such as the extra space of teh nugget? Let's put more technical matters here, so reviewers can appreciate why we have to crude partition the drive like this.)** Second, after the migration the previous region might need to be securely erased; here, not overlapping the two regions would make things more straightforward for erase secure. We are not aware of any solution that allow multiple ciphers activated per drive volume.

Selective Switching.

(HSG$_9$: I have a hard time reading this, considering that we want to map the case study of select files that we encrypt differently, *e.g.,* , encrypt legal files in a more secure eway. I think that case study doesn't really match selective switching, and actually I'm lost on the best example for selective switching. The reason is that if we want to say: we allow selective switching by allowing certain files to be configured different ways, then it means we should allow multiple configurations to

be active at the same time. Now we could still have only one configuration that is active, but what that means is that while I read the legal files, reading other files must be queued inside the FLEXCRYPT layer, because FLEXCRYPT only has one active configuration. So this will reduce the performance. If we want to make a scenario like this, we need to know why it is fundamentally impossible/hard to allow multiple cipher configurations to be active at the same time.)

(HSG$_{10}$: also, need to talk to Bernard, whether dividing up the partitions is really fundemental because of the way metadata/transcation journal is managed???)

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.

4.2 Generic Stream Cipher Interface

(HSG$_{11}$: page 5 starting)

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 [16]) 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.

... and interface for FLEXCRYPTS. Comes with an interface.

(HSG$_{12}$: what do you exactly mean by:
- "implementation detail"
- "cipher implementations"
- the "encryption/decryption process"
- "modification to third party code"
- "allows any stream cipher to be integrated to FLEXCRYPT"
- what is the "cryptographic driver"
- "interface" – interface between what and what? between the OS and FLEXCRYPT? between FLEXCRYPT and the cipher configuration? who calls the xorInterface, read/writeInterface?
- "FLEXCRYPT internals"
)

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 [17]. 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 [9]. HC-128 and other ciphers have similarly disparate requirements.

(HSG$_{13}$: need a figure to show what's going on)

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. **??**). 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.

`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.

`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.

(HSG$_{14}$: more examples needed here) And then with this interface, we have implemented, each in **xxx -xxx** LOC without the core cipher algorithm.

4.3 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. 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. To address this need, we propose a novel evaluation framework (see: Table 2). 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 2: **Quantifying ciphers.** *Our framework for classifying stream ciphers according to three ideal features: relative round count, ciphertext randomization, and ciphertext expansion, as discussed in Section* **??**.

Table **??** ..... 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).

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 [20]. 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 [11, 20].

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 [9]. 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 earlier in this work). 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" [16].

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 [9]. 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).

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 [9]. 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.

(HSG$_{15}$: So at the end how do we use this? what are these for? we need to give some examples here, or perhaps already covered in case studies in later sections??)

4.4 Putting It All Together

After describing the three main contributions, now we discuss other details surrounding the three main components.

Secure metadata management The focus of this paper is to implement mechanisms and policies to perform flexible switching of cipher configurations that can be built on top of existing block-level encryption module (aka. "encryption driver") that already provides the management of the encryption data structures. There were a couple of open-source choices to start with such as Linux `dm-crypt` [**?**], `cryptsetup` [**?**], `cryptmount` [**?**], or StrongBox [**?**], **(HSG$_{16}$: please double check i'm correct)**. We decided to build atop StrongBox because it already implements stream ciphers such as ChaCha which is more secure than block cipher.

FLEXCRYPT depends on several data structure management provided in StrongBox, such as its transcation and rekeying journal (for never writing data encrypted with the same key to the same location, hence avoiding pad reuse violations), Merkle tree (for tracking the drive state such as **(HSG$_{17}$: explain a bit more)**), monotonic counter (on a trusted hardware to prevent rollbacks), keycount store (to derive the nugget's unique encryption key from some master secret and limit the maximum length of any plaintext input to ciphers), and per-nugget metadata (to store cipher-specific extra metadata **(HSG$_{18}$: true??)**). Every drive partition also has a "head" area that indicates which cipher is currently active.

While we reuse some of the components, all of these are tightly integrated to the ciphers that they implemented (specifically ChaCha, and **xxx**). We had to untangle this, hence the contribution in the FLEXCRYPTS component where we now expose more structured interfaces allowing cipher implementors to natly build the metadata management of their cipher algorithm around the interfaces. More specifically, out of all the five **(HSG$_{19}$: true??)** StrongBox components above, only **xxx** can be reused as is, while the rest needs to be modularized and rewritten.

Threat model under switching

In terms of *confidentiality*, an adversary should not be able to reveal any information about encrypted plaintext without the proper key. As with prior works, 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. In terms of *data integrity*, an adversary should not be able to tamper with ciphertext and it go unnoticed. Nugget integrity is tracked by StrongBox's in-memory Merkle tree (see [16, Section **xxx**] 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. **(HSG$_{20}$: need to double check that this statement is still true, after I already rewrite the design section.)**

Higher-level integration FLEXCRYPT expects a higher-level integration/policy that will tell FLEXCRYPT what kind of switching should be performed and when. For example, for the battery-life scenario, FLEXCRYPT expects that the OS battery saver application will trigger the forward switching.

Cipher	Code origin
AES-XTS	OpenSSL 1.1.0h [**?**] and LibSodium 1.0.12 [**?**]
AES-CTR	OpenSSL 1.1.0h [**?**] and LibSodium 1.0.12 [**?**]
ChaCha	Floddyberry's ARM NEO [17]
Freestyle	Freestyle [**?**]
eSTREAM Profile 1	libestream cryptographic library [19]

Table 3: **Cipher code.** *The table shows the origin of the cipher code that we use but have to be repackaged in* FLEXCRYPT.

We provide more in the case studies section.

Generality FLEXCRYPT can be seen as a drop-in replacement for the popular Linux `dm-crypt` layer (encryption driver). For performance reasons, just like StrongBox, FLEXCRYPT recommends a log-based file system such as F2FS, **xxx** , or **xxx** , which are commonly used for flash devices. The reason for this is that supporting streaming chipers is more efficient in storage systems with no in-place updates. FLEXCRYPT is a software solution, however the same logic can be adopted to storage devices in the future, especially flash devices. For example, the no in-place update of the FTL nature will help stream chipers be performance while at the same time users can use other popular in-place update file systems such as ext4, btrfs, and xfs.

Switching modes pros/cons. **(HSG$_{21}$: DO WE STILL NEED THIS DISCUSSION? convergence, waste, performance, etc. THE FACT that the modes above are about case studies??? so it's known that there is no single best scenario.)**

5 Implementation

Our FLEXCRYPT implementation consists of **xxx** lines of C code (excluding StrongBox components that we re-use as is). To ensure high quality code, we also wrote 6,077 lines of ttest suite. All will be made open source to the public. We currently deploy FLEXCRYPT on top of the BUSE virtual block device [7] 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.

For the ciphers we suport, we select five types of ciphers for the purpose of this research: ChaCha8 and ChaCha20 [11], Freestyle [9] 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$)$. **(HSG$_{22}$: how about AES-XTRS and AES-CTR, and eSTREAM?? how should I reconcile these "5" ciphers with more of the ciphers listed in Table 3. I thought you support more than five??)** Table 3 shows the source of the cipher code that we use, but as mentioned in Section **??**, they must be repackaged with the FLEXCRYPTS interface.

(empty page)

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

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

[7] A block device in userspace, 2012.

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

[9] 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.

[10] 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.

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

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

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

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

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

[16] 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.

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

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

[19] Lvella. lvella/libestream.

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

[21] 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.

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

[23] 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.

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