# FLEXCRYPT: Kernel Support for Heterogeneous Full Drive Encryption

Paper #9

## 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 disk controllers/FTL and secure hardware [32, 63, 71, 72]; filesystems [11, 16, 24, 25, 33, 38, 70]; network storage and cloud storage [9, 19, 28, 36, 39, 41–43, 47, 48, 51, 53, 56, 59, 64, 67], de-duplication and secure erasure [17, 57, 65]; as well as kernels, databases, and other software [18, 45, 58, 62].

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). Popular FDE solutions include dm-crypt [1, 8] for Linux and BitLocker for Windows [26, 50]. Behind these implementations is the industry standard AES *block cipher* in XTS mode: AES-XTS [5, 6, 54]. Unfortunately, using AES-XTS introduces overhead that drastically impacts system performance and energy consumption. The story of FDE on Google's Android OS illustrates the problem. Android supported FDE with the release of Android 3.0, yet it was not enabled by default until Android 6.0 [21]. 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" [61]. 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 [34].

Fortunately, in the last two years there have been significant advancements in Full Drive Encryption; specifically, the slow AES block cipher can be replaced with fast *stream* ciphers like CHACHA20. For instance, constructions like Google's HBSH (hash, block cipher, stream cipher, hash)/Adiantum [20] and StrongBox [23] bring stream cipher based FDE to devices that do not or cannot support hardware accelerated AES, generally providing improved performance along the way [23]. A key to efficiently adopting stream ciphers into the storage layer is to pair it with a Log-structured File System (LFS) such as F2FS on flash devices. This is because LFSes naturally avoid writing to the same location multiple times, which requires an expensive re-keying operation when using stream cipher based FDE.

Traditionally, FDE at the block level requires enciphering drive contents *homogeneously* (with a single cipher), but these advancements reveal a new opportunity: storage systems that operate beyond the rigid constraints of prior work to support flexible *heterogeneous* FDE. Yet we find no storage system or OS that can support such a feature. To see where this might be useful, suppose the user of an mobile device is required to encrypt their drive with cipher FREESTYLE because it has the useful cryptographic property of being safe to back up to cloud storage. In exchange for this property, FREESTYLE uses a significant amount of energy when executing. Further suppose this device enters a critical battery state and the user wishes to conserve as much energy as possible. It would be beneficial if an energy-aware storage system could also enter a battery saving state. With heterogeneous FDE, such a flexible configuration is now achievable: such a system can temporarily switch out the battery-heavy FREESTYLE for the extremely energy efficient CHACHA20 when accessing data. Since data encrypted with CHACHA20 is not suitable for cloud backups, we pause cloud backups until we leave the critical battery state and switch ciphers again—saving even more energy.

We present FLEXCRYPT, to the best of our knowledge the *first* storage system to provide block level kernel support for heterogeneous FDE. 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. By supporting heterogeneous FDE, FLEXCRYPT permits cipher choice to be be viewed as a dynamically configurable parameter as opposed to a static choice made at format or boot time. As a result, FLEXCRYPT enables the storage system to adapt to changes that arise at runtime, including: resource availability and environment, desired security properties, and the OS's energy budget. This empowers users to perform cipher switching in space and time (*e.g.,* using more secure storage for more sensitive files, and/or dynamically switching between ciphers for one file), allowing the system to navigate the tradeoff space made by balancing competing security and latency requirements. FLEXCRYPT is composed primarily of three components that realize these benefits. These represent the three main contributions of this work.

First, we introduce FLEXSWITCH, a kernel configuration that exposes three switching models: *Forward*, *Selective* and *Mirrored*. Switching allows us to "re-cipher" storage units dynamically, enabling 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 by real case studies. For examples, Forward switching is motivated by the battery case study above; Selective switching is motivated by cases where users require certain files (*e.g.,* legal documents) to be stored more securely than others; and Mirrored switching is motivated by scenarios where system administrators want to change ciphers with zero downtime and without re-initializing the filesystem or device mapper.

Second, to support the switching models above, we implement FLEXAPI, a block-level module that contains encryption implementations ("crypts") that have been restructured to support switching. Prior work encrypts targets with a single cipher—*i.e.,* homogeneous FDE—where the choice of cipher implementation is integrated into the file/block layer system directly [8, 23]. The key challenge with supporting heterogeneous FDE—where multiple ciphers coexist on the same storage system—is that different ciphers take disparate inputs and produce disparate outputs. For example: CHACHA20's implementation requires a key and nonce, FREESTYLE's additionally requires configuration for output randomization, and AES-CTR's instead requires plaintext and sector information. At the same time, FREESTYLE and CHACHA20's implementations output a keystream that needs to be XORed with the plaintext to yield the ciphertext while AES-CTR outputs the ciphertext directly. Thus, we introduce a novel design substantively expanding prior FDE work by wholly decoupling cipher implementations from the encryption process. In FLEXAPI, we write hooks that effectively normalize the inputs and outputs required when switching between ciphers.

Finally, we present FLEXTRADE, a scheme that attempts to *quantify* the tradeoffs in the the rich configuration space of stream ciphers made available by the above. 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 conclude with a comprehensive evaluation of FLEXCRYPT. 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 7,048 as a kernel block module (will be open-sourced), and can act as an off-the-shelf replacement for dm-crypt. The ciphers are ChaCha [13], Freestyle [10], Salsa [14], AES in counter mode (AES-CTR) [4], Rabbit [15], Sosemanuk [12], HC-128 [37]. Second, we showcase the benefits of FLEXCRYPT with experiments illustrating three real-world case studies (*i.e.,* storage system responds to critical battery state, filesystem-agnostic file-level encryption, and downtime avoidance) and show the performance/energy tradeoffs. Finally, we perform several benchmarks to demonstrate the flexibility and performance of the individual ciphers and show the switching overhead of the three switching models we provide.

## 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.

### 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 [21, 34, 61]. 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 [23].

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, 60]. 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* ciphers. Block ciphers were a defacto solution for storage while stream ciphers were prevalent in networking **(HSG$_0$: true??)**. Their main differences are **(HSG$_1$: 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 ciphers full device encryption [20, 23] 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 in-place update nature of FTL or LFS-like storage allows the overwrite-heavy stream ciphers to be efficiently implemented (*e.g.,* ChaCha with F2FS or LogFS performs **xxx** times better than with ext4 [23]). 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-world 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 momentarily. 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 FLEXCRYPT 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 important to ensure the core logic of file systems does not have to be modified. Just like any other encryption technologies, 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 configurations 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 tightly integrated to the encrypted data and the on-disk data structures that the encryption layer uses. For this, we introduce the FLEXAPI (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.

## 3.1 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 "re-ciphering" must be done — using the just-deactivated 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

```
// 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             Re-ciphered on demand
         Re-ciphered 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: **Re-ciphering/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.

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 Table **??**.

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

Table 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$_2$: correct?)**. The reason we don't re-cipher 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 re-ciphered to $C_1$. For read operations, if the existing data (ciphered with $C_1$) is read during the switch, the data will be re-ciphered to $C_2$. The reason behind this is that re-ciphering 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 re-ciphered 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 re-ciphered 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 (supported 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 $C_2$.

During the migration, as summarized in Table 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 re-ciphered 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 [2, 3, 52], 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 fundamentally hard to have two cipher configurations enabled on the same drive. There are two reasons. First, ..**(HSG$_3$: 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.** 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.

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. FLEXCRYPT VSRs allows us to sidestep these issues.

**(HSG$_4$: also, need to talk to Bernard, whether dividing up the partitions is really fundemental because of the way metadata/transcation journal is managed???)**

When FLEXCRYPT 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 FLEXCRYPT 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. IO Differentiation is not new.

## 3.2 Generic Stream Cipher Interface

**Decoupling ciphers from the encryption process.** To flexibly switch between configurations in FLEXCRYPT 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 [23]) 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 FLEXAPI. Comes with an interface.

**(HSG$_5$: 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 FLEXCRYPT 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 FLEXCRYPT, 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 [27]. Using OpenSSL's AES implementation in CTR mode requires manually tracking the counter state and individual ciphertext blocks are retrieved though corresponding function invocations [55]. 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.

**(HSG$_6$: need a figure to show what's going on)**

Unlike prior work, FLEXCRYPT 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 FLEXCRYPT 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 FLEXCRYPT'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. FLEXCRYPT 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 FLEXCRYPT 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 Figure **??**). When a write request is received, the OS hands FLEXCRYPT 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 FLEXCRYPT 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. FLEXCRYPT 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 FLEXCRYPT 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 FLEXCRYPT 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.

And then with this interface, we have implemented **(HSG$_7$: examples of cipher wrappers implemented needed here)**, each in **xxx** - **xxx** LOC without the core cipher algorithm.

## 3.3 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 FLEXCRYPT 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. **[TODO: Combine what follows with what precedes.]** To address this need, we propose a novel evaluation framework (see: Table **??**). 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 Table 2 **[TODO: explanation]** . 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 [46]. 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 [13, 46].

**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 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" [23].

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).

**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.

**[TODO: Give an example here of how these would be used to judge a cipher and why.]**

## 3.4 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 [?], a-second-one [?], or StrongBox [23]. 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 transaction and rekeying journals (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$_8$: 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$_9$: 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 FLEXAPI component where we now expose more structured interfaces allowing cipher implementors to easily build the metadata management of their cipher algorithm around the interfaces. More specifically, out of all the five **(HSG$_{10}$: 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 [23, 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 FLEXCRYPT can be reduced to the individual confidentiality guarantees of the available ciphers used to encrypt nuggets. **(HSG$_{11}$: need to double check that this statement is still true, after I already rewrite the design section.)**

| Cipher | Source |
|---|---|
| AES | OpenSSL [55] & LibSodium [22] |
| ChaCha for ARM NEON [27] | |
| Freestyle | Freestyle [10] |
| eSTREAM Profile 1 | libestream cryptographic library [44] |

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

**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. 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 ciphers 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 ciphers be performance while at the same time users can use other popular in-place update file systems such as ext4, btrfs, and xfs.

## 4 Implementation

Our FLEXCRYPT implementation consists of 7,048 lines of C code (excluding StrongBox components we re-use as is). To ensure high quality code, we also wrote a 4,944 line test suite. Our implementation is available open-source. We 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), allowing our system to transact block layer requests in user space, reducing implementation complexity.

Among the many ciphers our implementation supports, we focus on five in this research: ChaCha8, ChaCha20 [13], and 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).

Table Table 3 shows the cipher implementations we use (repackaged with the FLEXAPI interface; see Subsection 3.2).

## 5 Evaluation

We implement FLEXCRYPT 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 FLEXCRYPT 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 FLEXCRYPT 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*. FLEXCRYPT is initialized at the primary configuration. Once we trigger a cipher switch, FLEXCRYPT 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 FLEXCRYPT'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).

To reason about the desirability of cipher configurations, we define the *security vector*, R+R, as a vector consisting of a configuration's relative rounds (Rounds) quantification and its ciphertext randomization (Randomization) quantification defined in Section 3. We define the norm over this vector as the sum of its components. Hence, we consider a cipher configuration "stronger" if it has higher ciphertext randomization and uses more rounds, *i.e.,* has a higher normed R+R.

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? (§5.1)
2. Can FLEXCRYPT achieve dynamic security/energy tradeoffs by reaching configuration points not accessible with prior work? (§5.2)
3. What is the performance and storage overhead of each cipher switching strategy? (§5.3)

## 5.1 Switching Strategies Under Various Workloads

### Baseline Cipher I/O Performance

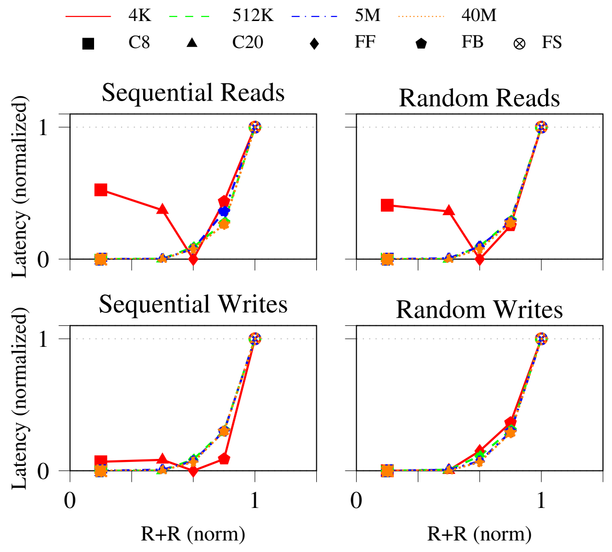

Figure 1: Median sequential and random write and then read performance baseline.

Figure 1 shows the normalized relative round count and ciphertext randomization (normed R+R vector) 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 overall latency and increased energy use for I/O operations. The relationship between these concerns is not always 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.

## 5.2 Reaching Between Static Configuration Points

### Forward Switching I/O Ratio Performance

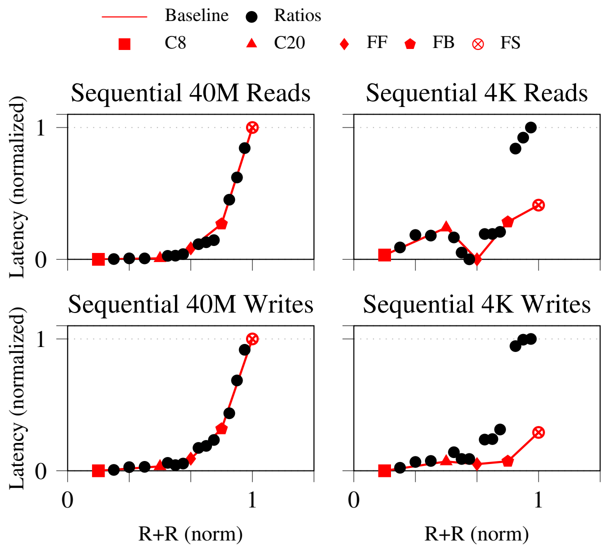

Figure 2: Median sequential and random write and then read back performance comparison of Forward switching to baseline. Each cluster of 3 dots between configurations represents the 7:3, 5:5, and 3:7 primary-vs-secondary I/O ratios described in the text.

Figure 2 shows the normalized relative round count and ciphertext randomization (normed R+R vector) 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 FLEXCRYPT 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 FLEXCRYPT can effectively transition the drive between ciphers without devastating performance. For the 40MB, 5MB, and 512KB workloads (40MB is shown), we see that FLEXCRYPT 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 Section **??**) and the fast completion time of the 4KB workloads preventing FLEXCRYPT 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 must be performed, and the algorithm itself is generally much slower than the ChaCha variants (see Figure 1). Doubly invoking Freestyle in a ratio configuration means these penalties are paid more often.

**Mirrored/Selective Switching I/O Ratio Performance**

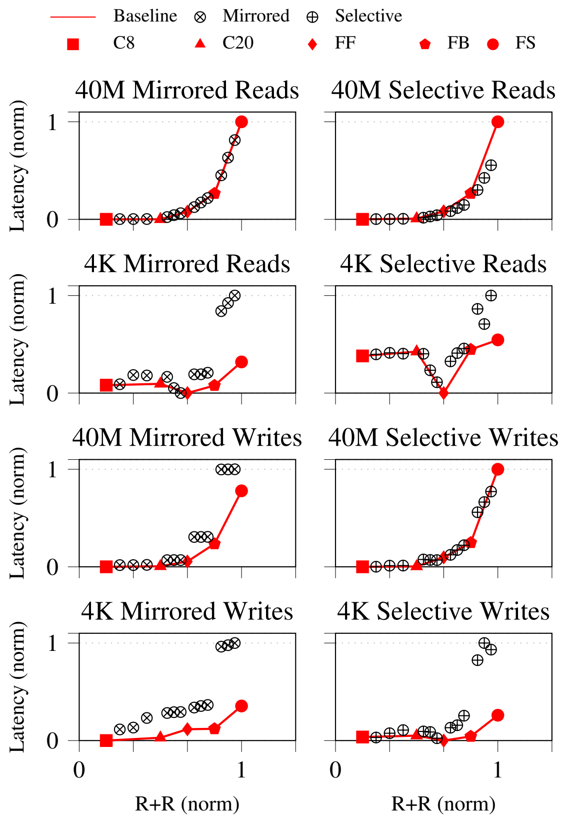

Figure 3: Median sequential and random write and then read back performance comparison of Mirrored and Selective switching strategies to baseline.

Figure 3 show the performance of the Mirrored and Selective strategies with the same configuration of ratios as Figure 2.

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 Figure 2, 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 is a tradeoff; Selective cannot quickly converge the drive to a single cipher configuration or survive the loss of an entire region.

## 5.3 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 FLEXCRYPT's amortization of cipher switching costs. Average overhead is 0.38x/0.44x for 4KB read/write workloads when FLEXCRYPT is unable to amortize cost. There is no spatial overhead with the Forward switching strategy.

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 FLEXCRYPT. FLEXCRYPT's design keeps these overheads acceptably low in practice, achieving the desired goal of flexibly navigating latency/security tradeoffs for FDE.

## 6 Case Studies

In this section, we provide fourxxx case studies demonstrating the practical utility of heterogeneous FDE not achievable with prior work. We cover a wide range of situations, highlighting concerns like meeting latency goals, trading off security and writable space, and keeping within an energy budget, all demonstrating the utility of both temporal and spatial switching strategies. All experiments are repeated multiple times.

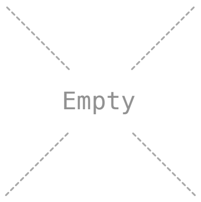

Figure 4: **Battery saver use case.** Energy-Security Tradeoff vs Strict Energy Budget as discussed in Section **??**. The graph shows median sequential write total energy use with respect to time with respect to time.

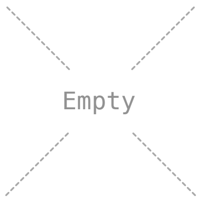

Figure 5: **Device end-of-life use case.** SSD EoL Use Case: Latency-Security Tradeoff vs Goals. Median sequential and random 40MB read and write performance comparison: baseline versus simulated faulty block device.

## 6.1 Battery Saver Mode

In this first case study, we revisit the motivational example. Here, the mobile device is "forced" to locally encrypt files with Freestyle Balanced Cipher ("$C_1$") for safely backing up the file to the backend cloud, but when the battery saver mode is on, the device will switch to a more energy-efficient cipher, ChaCha8 ("$C_2$"), and at the same time pausing the backup upload to the cloud momentarily (*e.g.,* the company does not want files to be saved in the cloud without $C_1$ encryption. Our goal is to complete I/Os as much as we can before the device dies.

To simulate I/O activity, we begin randomly writing 10 40MB files using the Freestyle Balanced cipher for 2 minutes. After the first 5 seconds, the device enters "battery saver" mode, which we simulate by underclocking the cores to their lowest frequencies and using `taskset` to transition FLEXCRYPT processes to the energy-efficient LITTLE cores. In this low battery mode, FLEXCRYPT switches to the ChaCha8 cipher.

**(HSG$_{12}$: I dont' understand Figure Figure 4. You said after 5 seconds. we switch to ChaCha8. But why the FB+ChaCha line is still going up? Why it doesn't flat out like the ChaCha line?? and why do even need to show the ChaCha8 only line?)**

**(HSG$_{13}$: we should exlude ChaCha8 only, unless you have a major point here)**

Figure 4 shows the 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. The figure shows two lines: **(1)** *Freestyle Balanced only*, that favors security even when backups are paused; the device dies before the I/Os complete. **(2)** *Freestyle Balanced + ChaCha8*, that performs the switch 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. When we get the device to a charger (not shown), the cloud backup is enabled again and when the nuggets are read, FLEXCRYPT automatically converges them back to Freestyle Balanced.

On average, using forward switching resulted in a 3.3x total energy reduction compared to exclusively using Freestyle Balanced, allowing us to remain within our energy budget. **(HSG$_{14}$: I don't see this 3.3x in the graph. Did you accidentally label the lines incorrectly?)**. We note, however, that the energy savings is not the point of this experiment. Rather, the lesson learned is that FLEXCRYPT 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.

## 6.2 End-of-Life Device Slowdown

Another usecase of forward switching is for offsetting the debilitating decline in performance when storage devices such as SSDs reach end-of-life (EoL); due to garbage collection and wear-leveling requirements of SSDs, as free space becomes constrained, I/O performance drops precipitously [31]. Let us suppose in this case, the system running on the dying SSDs have a strict latency budget (as long as meeting some minimum security requirements). The strict latency ceiling can be violated if the device latency increases. If FLEXCRYPT 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.

To show this, we begin by writing 10 40MB files per each cipher as a baseline. **(HSG$_{15}$: I don't understand "per each cipher")** We then introduce a delay of 20*ms* simulating drive slowdown. **(HSG$_{16}$: Where do you get "20ms" from? that's such a high latency. Any citation you can use here to back that up)**, **(HSG$_{17}$: These sentences below are non informative. You should help reviewers walk through the graph. I don't even understand what the x-axis in the Figure means. The best way to describe a figure, is describe the x-axis and then y-axis, and then pick two extreme points in the graph, and describe those points, what the values mean, so that reviewers can really understand them. if reviewers understand two extreme points, then they will undersstand the rest. Please redo the sentences.)** ~~In Figure 5, 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~~

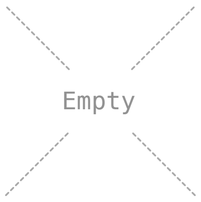

Figure 6: **Selective file/region usecase.** VSR Use Case: ChaCha8 vs Freestyle Strong Sequential 4KB, 5MB Performance. The graph shows median sequential read and write performance comparison of 4KB, 5MB I/O with 7-to-3 ratio, ChaCha8 vs Freestyle Strong (respectively).

~~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.~~

### 6.3 Select Data Encryption

This usecase illustrates utility of selective switching to achieve a performance win—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, while the rest can use a minimalist encryption. As mentioned before, this feature requires users to annotate certain files with a special tag via file system calls, which would then be stored in the inode. Because the FLEXCRYPT layer is file oblivious, every block I/O through the FLEXCRYPT layer will be labeled with the corresponding cipher.

We begin by with 10 5MB and 4KB write-read operations to two FLEXCRYPT drive instances: one using ChaCha8 and the other using Freestyle Strong. They represent two extreme where ChaCha8 is a low-latency cipher and Freestyle Strong exhibits a very high overhead. We then run another FLEXCRYPT instance with a 7:3 ratio where 30% of the data considered highly sensitive uses Freestyle Strong.

The left-side and right-side bars in Figure 6 show the two extremes; ChaCha8 exhibits a low latency, consistently less than 1 second across the workloads, while Freestyle Strong exhibits 5 to 20 second latency depending on the workload. The selective switching (middle bars) shows 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. Thus, selective switching keeps our sensitive data at the mandated security level while keeping the performance (and also battery life) benefits of using a fast cipher for the majority of I/O operations.

### 6.4 No-Downtime Encryption Upgrade

In Figure 7 [TODO: finish sentence] . **(HSG$_{18}$: we must have a short case study here. Otherwise no point of talking about mirrored switch)**.

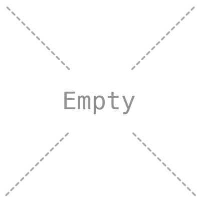

Figure 7: **Fix-me-todo.** [TODO: This.]

## 7 Related Work

We break the related work into three(or more**xxx** ) categories. **xxx** , **xxx** , and **xxx** . [TODO: Do this.]

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 [23], 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 [20]), and a stream cipher in a binary additive (XOR) mode leveraging LFS overwrite-averse behavior to prevent overwrites (StrongBox [23]).

Unlike StrongBox and other work, which focuses on optimizing performance despite re-ciphering due to overwrites, FLEXCRYPT 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 [29, 30, 40, 49, 66, 68, 69]. Goodman et al. introduced selectively decreasing the security of some data to save energy [29]. However, their approach is designed for communication and only considered iteration/round count, thus it did not anticipate the need for FLEXCRYPT's generic interface, switching strategies, or quantification framework. Wolter and Reinecke study approaches to quantifying security in several contexts [66]. 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 [20]. While not an FDE solution, LastPass has dealt with scaling the number of iterations of PBKDF#2, trading performance for security during login sessions [35].

On mobile energy/battery life, prior works have shown the importance of energy saving mode and bugs that drain energy [**?**, **?**].

## 8 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??.

**(HSG$_{19}$: Other notes:**

**- double check figure/table captions to be consistent to what the text says**

**- Add "as explained in Section ??" in every table/figure caption.**

**- Feel free to change any wording that I have. As mentioned before, I focus on the logical flow, not choice of words**

**- Keep things simple, don't introduce too many acronyms. E.g. I remove that "VSR" thing because it's selective basically**

**- Make sure figure placement is perfect. Put figures on the same page where they are being described. That's why I put all figure in separate fig-*.tex files, so you can easily move them around. Figure should always use [t]. Reviewers prefer that than figures that breaks the flow of the text**

**- Make sure figure captions explain the x-axis and y-axis properly. Do explicitly state "the x-axis shows ..., the y-axis shows ...". Also as I mentioned last time, do put more annotations inside the figure/graph to help reviewers read the final conclusion.)**

## References

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

[2] Bios-enabled security features in hp business notebooks.

[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] Marcos Kawazoe Aguilera, Minwen Ji, Mark Lillibridge, John MacCormick, Erwin Oertli, David G. Andersen, Michael Burrows, Timothy P. Mann, and Chandramohan A. Thekkath. Block-level security for network-attached disks. In Jeff Chase, editor, *Proceedings of the FAST '03 Conference on File and Storage Technologies, March 31 - April 2, 2003, Cathedral Hill Hotel, San Francisco, California, USA*. USENIX, 2003.

[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] Maurice Bailleu, Jörg Thalheim, Pramod Bhatotia, Christof Fetzer, Michio Honda, and Kapil Vaswani. SPEICHER: securing LSM-based key-value stores using shielded execution. In Arif Merchant and Hakim Weatherspoon, editors, *17th USENIX Conference on File and Storage Technologies, FAST 2019, Boston, MA, February 25-28, 2019*, pages 173–190. USENIX Association, 2019.

[12] 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.

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

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

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

[16] Jeff Bonwick and Bill Moore. ZFS: The last word in file systems.(2007). *URL http://www. opensolaris. org/os/community/zfs/docs/zfs_last. pdf*, 2007.

[17] Fabiano C. Botelho, Philip Shilane, Nitin Garg, and Windsor Hsu. Memory efficient sanitization of a deduplicated storage system. In Keith A. Smith and Yuanyuan Zhou, editors, *Proceedings of the 11th USENIX conference on File and Storage Technologies, FAST 2013, San Jose, CA, USA, February 12-15, 2013*, pages 81–94. USENIX, 2013.

[18] Adam Chlipala. Static checking of dynamically-varying security policies in database-backed applications. In Remzi H. Arpaci-Dusseau and Brad Chen, editors, *9th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2010, October 4-6, 2010, Vancouver, BC, Canada, Proceedings*, pages 105–118. USENIX Association, 2010.

[19] Landon P. Cox and Brian D. Noble. Samsara: honor among thieves in peer-to-peer storage. In Michael L. Scott and Larry L. Peterson, editors, *Proceedings of the 19th ACM Symposium on Operating Systems Principles 2003, SOSP 2003, Bolton Landing, NY, USA, October 19-22, 2003*, pages 120–132. ACM, 2003.

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

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

[22] Frank Denis. Libsodium 1.0.12.

[23] 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.

[24] Thanh Do, Tyler Harter, Yingchao Liu, Haryadi S. Gunawi, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. HARDFS: hardening HDFS with selective and lightweight versioning. In Keith A. Smith and Yuanyuan Zhou, editors, *Proceedings of the 11th USENIX conference on File and Storage Technologies, FAST 2013, San Jose, CA, USA, February 12-15, 2013*, pages 105–118. USENIX, 2013.

[25] Vandeir Eduardo, Luis Carlos Erpen De Bona, and Wagner M. Nunan Zola. Speculative encryption on GPU applied to cryptographic file systems. In Arif Merchant and Hakim Weatherspoon, editors, *17th USENIX Conference on File and Storage Technologies, FAST 2019, Boston, MA, February 25-28, 2019*, pages 93–105. USENIX Association, 2019.

[26] Niels Ferguson. AES-CBC+ elephant diffuser: A disk encryption algorithm for windows vista. 2006.

[27] Floodyberry. floodyberry/chacha-opt, March 2015.

[28] Kevin Fu, M. Frans Kaashoek, and David Mazières. Fast and secure distributed read-only file system. In Michael B. Jones and M. Frans Kaashoek, editors, *4th Symposium on Operating System Design and Implementation (OSDI 2000), San Diego, California, USA, October 23-25, 2000*, pages 181–196. USENIX Association, 2000.

[29] 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.

[30] 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, October 2007.

[31] 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.

[32] Pieter H. Hartel, Leon Abelmann, and Mohammed G. Khatib. Towards tamper-evident storage on patterned media. In Mary Baker and Erik Riedel, editors, *6th USENIX Conference on File and Storage Technologies, FAST 2008, February 26-29, 2008, San Jose, CA, USA*, pages 283–296. USENIX, 2008.

[33] Ragib Hasan, Radu Sion, and Marianne Winslett. The case of the fake picasso: Preventing history forgery with secure provenance. In Margo I. Seltzer and Richard Wheeler, editors, *7th USENIX Conference on File and Storage Technologies, February 24-27, 2009, San Francisco, CA, USA. Proceedings*, pages 1–14. USENIX, 2009.

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

[35] LogMeIn Inc. What makes lastpass secure? logmein support entry.

[36] Michael Kaminsky, George Savvides, David Mazières, and M. Frans Kaashoek. Decentralized user authentication in a global file system. In Michael L. Scott and Larry L. Peterson, editors, *Proceedings of the 19th ACM Symposium on Operating Systems Principles 2003, SOSP 2003, Bolton Landing, NY, USA, October 19-22, 2003*, pages 60–73. ACM, 2003.

[37] Ayesha Khalid, Prasanna Ravi, Goutam Paul, and Anupam Chattopadhyay. One word/cycle HC-128 accelerator via state-splitting optimization. 2014.

[38] Changman Lee, Dongho Sim, Joo Young Hwang, and Sangyeun Cho. F2FS: A new file system for flash storage. In Jiri Schindler and Erez Zadok, editors, *Proceedings of the 13th USENIX Conference on File and Storage Technologies, FAST 2015, Santa Clara, CA, USA, February 16-19, 2015*, pages 273–286. USENIX Association, 2015.

[39] Jinyuan Li, Maxwell N. Krohn, David Mazières, and Dennis E. Shasha. Secure untrusted data repository (SUNDR). In Eric A. Brewer and Peter Chen, editors, *6th Symposium on Operating System Design and Implementation (OSDI 2004), San Francisco, California, USA, December 6-8, 2004*, pages 121–136. USENIX Association, 2004.

[40] 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.

[41] Yan Li, Nakul Sanjay Dhotre, Yasuhiro Ohara, Thomas M. Kroeger, Ethan L. Miller, and Darrell D. E. Long. Horus: fine-grained encryption-based security for large-scale storage. In Keith A. Smith and Yuanyuan Zhou, editors, *Proceedings of the 11th USENIX conference on File and Storage Technologies, FAST 2013, San Jose, CA, USA, February 12-15, 2013*, pages 147–160. USENIX, 2013.

[42] Jed Liu, Michael D. George, K. Vikram, Xin Qi, Lucas Waye, and Andrew C. Myers. Fabric: a platform for secure distributed computation and storage. In Jeanna Neefe Matthews and Thomas E. Anderson, editors, *Proceedings of the 22nd ACM Symposium on Operating Systems Principles 2009, SOSP 2009, Big Sky, Montana, USA, October 11-14, 2009*, pages 321–334. ACM, 2009.

[43] Jacob R. Lorch, Bryan Parno, James W. Mickens, Mariana Raykova, and Joshua Schiffman. Shroud: ensuring private access to large-scale data in the data center. In Keith A. Smith and Yuanyuan Zhou, editors, *Proceedings of the 11th USENIX conference on File and Storage Technologies, FAST 2013, San Jose, CA, USA, February 12-15, 2013*, pages 199–214. USENIX, 2013.

[44] Lvella. lvella/libestream.

[45] Umesh Maheshwari, Radek Vingralek, and William Shapiro. How to build a trusted database system on untrusted storage. In Michael B. Jones and M. Frans Kaashoek, editors, *4th Symposium on Operating System Design and Implementation (OSDI 2000), San Diego, California, USA, October 23-25, 2000*, pages 135–150. USENIX Association, 2000.

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

[47] Petros Maniatis and Mary Baker. Enabling the archival storage of signed documents. In Darrell D. E. Long, editor, *Proceedings of the FAST '02 Conference on File and Storage Technologies, January 28-30, 2002, Monterey, California, USA*, pages 31–45. USENIX, 2002.

[48] Michelle L. Mazurek, Yuan Liang, William Melicher, Manya Sleeper, Lujo Bauer, Gregory R. Ganger, Nitin Gupta, and Michael K. Reiter. Toward strong, usable access control for shared distributed data. In Bianca Schroeder and Eno Thereska, editors, *Proceedings of the 12th USENIX conference on File and Storage Technologies, FAST 2014, Santa Clara, CA, USA, February 17-20, 2014*, pages 89–103. USENIX, 2014.

[49] 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.

[50] Micrsoft. What's new in windows 10, versions 1507 and 1511 (windows 10) - what's new in windows.

[51] Ethan L. Miller, Darrell D. E. Long, William E. Freeman, and Benjamin Reed. Strong security for network-attached storage. In Darrell D. E. Long, editor, *Proceedings of the FAST '02 Conference on File and Storage Technologies, January 28-30, 2002, Monterey, California, USA*, pages 1–13. USENIX, 2002.

[52] 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.

[53] Athicha Muthitacharoen, Robert Tappan Morris, Thomer M. Gil, and Benjie Chen. Ivy: A read/write peer-to-peer file system. In David E. Culler and Peter Druschel, editors, *5th Symposium on Operating System Design and Implementation (OSDI 2002), Boston, Massachusetts, USA, December 9-11, 2002*. USENIX Association, 2002.

[54] NIST. Public comments on the XTS-AES mode, 2008.

[55] OpenSSL. Openssl 1.1.0h.

[56] Antonis Papadimitriou, Ranjita Bhagwan, Nishanth Chandran, Ramachandran Ramjee, Andreas Haeberlen, Harmeet Singh, Abhishek Modi, and Saikrishna Badrinarayanan. Big data analytics over encrypted datasets with seabed. In Kimberly Keeton and Timothy Roscoe, editors, *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*, pages 587–602. USENIX Association, 2016.

[57] Zachary N. J. Peterson, Randal C. Burns, Joseph Herring, Adam Stubblefield, and Aviel D. Rubin. Secure deletion for a versioning file system. In Garth Gibson, editor, *Proceedings of the FAST '05 Conference on File and Storage Technologies, December 13-16, 2005, San Francisco, California, USA*. USENIX, 2005.

[58] Raluca A. Popa, Catherine M. S. Redfield, Nickolai Zeldovich, and Hari Balakrishnan. CryptDB: protecting confidentiality with encrypted query processing. In Ted Wobber and Peter Druschel, editors, *Proceedings of the 23rd ACM Symposium on Operating Systems Principles 2011, SOSP 2011, Cascais, Portugal, October 23-26, 2011*, pages 85–100. ACM, 2011.

[59] Jason K. Resch and James S. Plank. AONT-RS: blending security and performance in dispersed storage systems. In Gregory R. Ganger and John Wilkes, editors, *9th USENIX Conference on File and Storage Technologies, San Jose, CA, USA, February 15-17, 2011*, pages 191–202. USENIX, 2011.

[60] 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.

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

[62] Richard P. Spillane, Sachin Gaikwad, Manjunath Chinni, Erez Zadok, and Charles P. Wright. Enabling transactional file access via lightweight kernel extensions. In Margo I. Seltzer and Richard Wheeler, editors, *7th USENIX Conference on File and Storage Technologies, February 24-27, 2009, San Francisco, CA, USA. Proceedings*, pages 29–42. USENIX, 2009.

[63] John D. Strunk, Garth R. Goodson, Michael L. Scheinholtz, Craig A. N. Soules, and Gregory R. Ganger. Self-securing storage: Protecting data in compromised systems. In Michael B. Jones and M. Frans Kaashoek, editors, *4th Symposium on Operating System Design and Implementation (OSDI 2000), San Diego, California, USA, October 23-25, 2000*, pages 165–180. USENIX Association, 2000.

[64] Kaushik Veeraraghavan, Andrew Myrick, and Jason Flinn. Cobalt: Separating content distribution from authorization in distributed file systems. In Andrea C. Arpaci-Dusseau and Remzi H. Arpaci-Dusseau, editors, *5th USENIX Conference on File and Storage Technologies, FAST 2007, February 13-16, 2007, San Jose, CA, USA*, pages 231–244. USENIX, 2007.

[65] Michael Yung Chung Wei, Laura M. Grupp, Frederick E. Spada, and Steven Swanson. Reliably erasing data from flash-based solid state drives. In Gregory R. Ganger and John Wilkes, editors, *9th USENIX Conference on File and Storage Technologies, San Jose, CA, USA, February 15-17, 2011*, pages 105–117. USENIX, 2011.

[66] 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.

[67] Aydan R. Yumerefendi and Jeffrey S. Chase. Strong accountability for network storage. In Andrea C. Arpaci-Dusseau and Remzi H. Arpaci-Dusseau, editors, *5th USENIX Conference on File and Storage Technologies, FAST 2007, February 13-16, 2007, San Jose, CA, USA*, pages 77–92. USENIX, 2007.

[68] 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.

[69] 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, February 2013.

[70] Yupu Zhang, Abhishek Rajimwale, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. End-to-end data integrity for file systems: A ZFS case study. In Randal C. Burns and Kimberly Keeton, editors, *8th USENIX Conference on File and Storage Technologies, San Jose, CA, USA, February 23-26, 2010*, pages 29–42. USENIX, 2010.

[71] Tianli Zhou and Chao Tian. Fast erasure coding for data storage: A comprehensive study of the acceleration techniques. In Arif Merchant and Hakim Weatherspoon, editors, *17th USENIX Conference on File and Storage Technologies, FAST 2019, Boston, MA, February 25-28, 2019*, pages 317–329. USENIX Association, 2019.

[72] Aviad Zuck, Yue Li, Jehoshua Bruck, Donald E. Porter, and Dan Tsafrir. Stash in a flash. In Nitin Agrawal and Raju Rangaswami, editors, *16th USENIX Conference on File and Storage Technologies, FAST 2018, Oakland, CA, USA, February 12-15, 2018*, pages 169–188. USENIX Association, 2018.