FLEXCRYPT: Kernel Support for Heterogeneous Full Drive Encryption

Paper #67

Abstract

[TODO:
This.
This.
This.
This.
This.
This.
This.
This.
This.
This.
This.
This.
This.
This.]

1 Introduction

Security is an essential 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 [38,82,91,93]; filesystems [13,18,28,29,39,49,90]; network storage and cloud storage [10,22,33,45,52,54,56, 57,62,63,67,70,73,76,84,87], de-duplication and secure erasure [19,74,85]; as well as kernels, databases, and other software [20,60,75,81].

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 [30,66]. Behind these implementations is the industry standard AES *block cipher* in XTS mode: AES-XTS [5,6,71]. 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 [24]. 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" [80]. 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 [41].

Fortunately, recent work has produced significant advancements in Full Drive Encryption. Specifically, the slow AES block cipher can be replaced with fast *stream* ciphers like CHACHA20. For instance, both Google's HBSH (hash, block cipher, stream cipher, hash)/Adiantum [23] and StrongBox [27] bring the performance benefits of stream cipher based FDE to devices that do not or cannot support hardware accelerated AES [27]. 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 [50] 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 the stream-cipher based approaches 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 a mobile device user 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, FREESTYLE uses significant 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: it can temporarily switch out the energy-inefficient FREESTYLE for the extremely energy-efficient CHACHA20 when accessing data. Since data encrypted with CHACHA20 is not suitable for cloud backups, backups are paused until we leave the critical battery state and switch ciphers again—saving even more energy.

To realize this flexibility, we present FLEXCRYPT, to the best of our knowledge the *first* storage system to provide block level kernel support for heterogeneous FDE. Different ciphers expose different performance, energy, and security 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 runtime changes 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*, *Mirrored* and *Selective*. 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 example, Forward switching is motivated by the battery case study above; 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; and Selective switching is motivated by cases where users require certain files (*e.g.,* legal documents) to be stored more securely than others.

Second, to support the switching models above, we implement FLEXAPI, a block layer kernel 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 [8,27]. 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, an framework that evaluates crypts based on concerns important in the context of heterogeneous FDE. Using this framework, we expose a rich tradeoff space of crypts over three competing concerns: read/write performance (latency) and total energy use, desirable security properties, and total writable space on the drive. FLEXTRADE gives non-cryptographers a quick and intuitive method to reason about which crypt to use when.

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 lines of code as a kernel block module [1], and can act as an off-the-shelf replacement for dm-crypt. The ciphers are ChaCha [15], Freestyle [12], Salsa [16], AES in counter mode (AES-CTR) [4], Rabbit [17], Sosemanuk [14], HC-128 [46]. Second, we showcase the benefits of FLEXCRYPT with experiments illustrating three real-world case studies (*i.e.,* storage system responds to critical battery state, datacenter downtime avoidance, and filesystem-agnostic file-level encryption) 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

In this section we discuss recent advancements and limitations in stream cipher based Full Drive Encryption (FDE), then provide three case studies that motivate our work.

2.1 The Homogeneous FDE Problem

The major issue with FDE is its homogeneous nature; users must commit to one FDE storage configuration at initialization time [8]. Suppose we have an ultra-low-voltage netbook provided by our employer who requires that the drive is 1) fully encrypted at all times and 2) backed up to an offsite system whenever possible. Given that the FDE industry standard is AES-XTS, we consider initializing our drive with it. At this point, 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 [24, 41, 80]. As the drive must be 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 will be degraded even further; I/O latency can be as high as 3–5x [27]. To provide the security without the overhead, there are *faster, more energy-efficient* stream ciphers we might consider instead.

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. 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, 77]. Unfortunately, the backups required by this scenario function as snapshots for an attacker. There are *ciphertext randomizing* stream ciphers we can use that can

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

be safely backed up, but they are known to be an additional source of latency and power overhead [12].

Third, our device 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 storage system should respond to these changing requirements, including reducing energy use when the user wants to conserve power (i.e. "battery saver mode"), all without violating any other concerns.

The fundamental challenge is that there is no single cipher that addresses all concerns simultaneously; however, these concerns change over time (e.g., energy efficieny is more important when battery is low). Hence, a homogeneous FDE solution will always sacrifice one concern for the other where a flexible *heterogeneous* approach may not.

2.2 Recent Advancements

Broadly speaking, an FDE storage system can be implemented in two ways: using *block* ciphers or using *stream* ciphers. Block ciphers were the de facto solution for storage while stream ciphers were prevalent in networking and elsewhere. The relevant difference: encryption using a stream cipher is simpler and thus faster than a block cipher, especially in software; however, stream ciphers have their own non-trivial problems (*i.e.,* rollback attacks and overwrites) and are hence not a drop-in replacement for block ciphers [27]. There have been major advancements in the FDE technology that moves us away from slower block ciphers and towards the adoption of faster stream cipher based FDE [23, 27].

There are two technological shifts that enable secure, 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) [47, 50, 78]. The no in-place update "overwrite-averse" nature of FTL and LFS-like storage allows stream cipher based FDE to be efficiently implemented (*e.g.,* ChaCha20 with F2FS performs on average 1.72× better than ext4 [27]). Second, mobile devices now support trusted hardware, such as Trusted Execution Environments (TEE) [55, 83] and secure storage areas [9], which means the system 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

Recent advancements demonstrate that stream cipher based FDE can be implemented in an efficient manner, which opens up an opportunity for the storage layer to support multiple encryption technologies in various modes and configurations. For instance, such a system could switch from using an energy-inefficient but backup-safe cipher to an energy-efficient cipher to save battery life when entering battery saver mode and later switch back (and resume uploading backups) after leaving battery-saver mode. Clearly, there is no one-size-fits-all encryption solution and some flexibility is demanded. Below we present three such case studies.

"Battery saver" mode and Forward switching. In situations like the netbook example above, the user might want to prioritize reducing the total energy use when the battery is low. While in this "battery saver mode," it would be beneficial if the storage system could switch to a more energy-efficient cipher and pause the cloud backup momentarily. A user would be willing to accept this tradeoff knowing that within a few hours they will have access to power and could resume uploads when the storage system switches back, which is no different from an internet connection problem. This case is pervasive [11, 35, 36, 51, 65].

No-downtime encryption upgrade and Mirrored switching. From mobile devices, we now turn to server-side storage where we might require an encryption upgrade [21, 40, 43]. A server provider might decide to completely upgrade from one encryption technology (*e.g.,* that has been superseded) to another. Ideally, during the switch, the server would maintain its service-level agreements without any restarts or downtime. However, without kernel-level support, the server provider must write application-level software that performs the whole switching operation and manually redirects users to the appropriate files. It would be beneficial if this could be handled entirely at the storage layer instead.

Scalable encryption and Selective switching. It is known that systems communicating over a network transmit both high- and low-priority data and those priority levels should be encrypted differently [34]. It would be beneficial if our storage system could take advantage of the heterogeneous block-by-block "scalable" nature of such an encryption requirement to reduce latency and conserve energy. Examples of data with variable encryption requirements include corporate and government documents where information is routinely classified/redacted, credit card or information in a shopping transaction [34], and directory or file-level encryption transparent to the filesystem.

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, when using the Selective switching model, only a small modification in the file system layer to include cipher choice in writes to the block layer. The choice to implement FLEXCRYPT at the block layer (as dm-crypt [8] does) is important to ensure the core logic of file systems does not have to be modified and that all metadata is encrypted and protected. Just like any other encryption technology, FLEXCRYPT must have a unit of encryption/decryption. In this paper we call that unit a "nugget," which can be configured as one or more sequential blocks.

To provide kernel support for heterogeneous FDE and live cipher switching, we must address three key challenges:

1. We must provide switching models that cover the temporal and spatial nature of storage access to support the flexible heterogeneous encryption that users demand. For this, we introduce the FLEXSWITCH (§3.1) component of FLEXCRYPT that exposes three switching models to users: Forward, Selective, and Mirrored switching.
2. We must allow arbitrary cipher implementations to be easily integrated into FLEXCRYPT. This is non-trivial, however, as different ciphers and encryption modes require different inputs, generate disparate outputs, and are often tightly coupled to the storage/encryption layer itself. For this, we introduce the FLEXAPI (§3.2) component where we decouple cipher implementations from the core encryption algorithm by providing wrapper interfaces that allow different encryption algorithms ("crypts") to co-exist in the storage layer.
3. Most users do not care about this crypt or that crypt, they just want a solution that does what they want; hence, we must help users quickly and intuitively understand the tradeoffs between different crypts. To facilitate this, we introduce FLEXTRADE (§3.3), an evaluation framework that scores crypts according to three properties important in the context of heterogeneous FDE: a crypt's round count (relative to other crypts of the same cipher implementation), if a crypt randomizes its output, and if a crypt generates more output bytes than input bytes.

3.1 FLEXSWITCH: Cipher Switching Models

Though not a technical limitation, it is helpful to think of FLEXCRYPT as having a single *active cipher configuration* among several available. With FLEXSWITCH, FLEXCRYPT has the flexibility to switch from one cipher configuration to another, or even employ multiple configurations in a way unachievable with prior work. When a cipher switch is triggered, a different configuration becomes the active configuration, and "re-ciphering" must be done; *i.e.,* using the deactivated configuration to decrypt a nugget's contents and using the active configuration to re-cipher it.

The challenge here is to accomplish re-ciphering while minimizing overhead. A naive approach would switch every nugget to the active configuration immediately, but the latency and energy cost would be unacceptable. Hence, a more strategic approach is necessary. 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. These models allow for nuggets to be re-ciphered in a variety of cases with minimal impact on performance and energy but without compromising security; they are: Forward, Mirrored, and Selective. How FLEXCRYPT reacts to I/Os under each model is summarized in table 1.

	Read	**Write**
Forward	1st read: D_{C_1} then E_{C_2} Nth read: D_{C_2} Or variant version: 1st read: D_{C_1} 2nd read: D_{C_1} then E_{C_2} Nth read: D_{C_2}	E_{C_2} (only if necessary)
Mirrored	When migrating: D_{C_1} from C_1's region After migrating: D_{C_2} from C_2's region	I/O is duplicated: E_{C_1} to C_1's region and E_{C_2} to C_2's region
Selective	Encrypted with C_1: D_{C_1} Encrypted with C_2: D_{C_2}	Should use C_1: E_{C_1} Should use C_2: E_{C_2}

Table 1: **Switching Models and Re-ciphering.** This table summarizes for each switching model what happens on I/Os to a nugget when FLEXCRYPT switches from cipher C_1 to cipher C_2 (see §3.1). "Read" denotes returning data during the switch. "Write" denotes committing new data. E_X denotes encryption (*i.e.,* re-ciphering) and D_X denotes decryption using cipher X.

Forward switching. In the case of the battery saver mode example, we want to switch from cipher "C_1", a highly-secure, energy-expensive cipher to cipher "C_2," a less-secure but more energy-efficient cipher. However, for efficiency, *we only switch nuggets on demand*; *i.e.,* we change those nuggets that are touched during I/O. This switching model would be enabled as part of a higher-level policy (*i.e.,* battery saver mode) enacted by the OS or user.

Table 1 summarizes what happens on I/Os immediately after the active cipher is switched to C_2. For nugget-sized writes, new data is ciphered with C_2 and committed as normal. For intra-nugget writes, the nugget is first decrypted using C_1 and then re-ciphered using C_2 with the new data included. The new nugget is then committed as normal. Re-ciphering only occurs when necessary; once a nugget is ciphered with C_2, no further switching is required. For reads, the nugget is decrypted using C_1, it is re-ciphered with C_2, and then the data is returned. On subsequent reads, the nugget is decrypted with C_2 and read as normal.

Later, when battery saver mode is turned off and C_1 becomes the active cipher again, nuggets are not automatically re-ciphered as, again, that would be inefficient. Nuggets will be re-ciphered on demand using the converse of the process described above. Thus, Forward switching's performance and battery impact is limited to the 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 model. For instance, a variation on read operations: we consider the first, second, and nth read accesses distinctly. On the first read, the nugget is decrypted using C_1 and the data returned. Only if that same nugget is read a second

time is it re-ciphered with C_2 before the data is returned. On subsequent reads, the nugget is decrypted with C_2 and read as normal. The intuition is that Forward switching only happens temporarily (*e.g.,* while the battery is low), hence data being read might only be read once and stay in memory. However, if the nugget is read the second time, then the data is re-ciphered to C_2. This and other possible variants are left for future work. So far, we find our version of Forward switching suits a common battery-saving scenario.

Mirrored switching. Next, we consider a different scenario where "C_1" is a cipher that has been superseded by "C_2", a newly recommended cipher that has just been added to FLEXCRYPT's latest kernel/module upgrade. Let us imagine a datacenter storage operator who wants to upgrade from C_1 to C_2 without any service interruptions. Anticipating this, our operator partitions available storage into two regions. That is, to support a full drive cipher switching supported by the block layer and without application/file system modification, we must pay the space cost to anticipate such a switch.

During the migration, as summarized in table 1, all write operations that hit C_1's region will be mirrored to C_2's region. Meanwhile, in the background, the data in C_1's region is incrementally re-ciphered and written to C_2's region. Until the migration is complete, read operations will be served by C_1's region. This is very similar to VM live migration [79]. After the migration completes, one can use a mechanism such as SSD Instant/Secure Erase [2, 3, 69] to securely erase the previous storage region's data.

In general, Mirrored switching allows us to converge a drive volume to a single cipher configuration without losing any data, suffering outages or downtime, requiring onerous manual effort, or violating any critical service-level agreements. We note that, after the migration, the previous region's data needs to be securely erased to complete the upgrade; here, not overlapping the two regions makes Instant/Secure Erase straightforward [2, 3].

Selective switching. Finally, we consider the case of "scalable encryption." The key insight lies in recognizing that, when storing classified materials, corporate secrets, intellectual property, and other information that requires the highest level of discretion, such sensitive information often appears within a (much) larger amount of data that is valued less. For example, perhaps banking transaction information is littered throughout a PDF; perhaps passwords and other sensitive information exists within several much larger log files. Using prior techniques, either all the data would be stored with high-overhead cipher, the critical data would be stored without the mandated cipher type, or the data would have to be split among separate files requiring a complex and potentially error-prone encryption management scheme.

FLEXCRYPT allows us to sidestep these issues by supporting heterogeneous FDE, *i.e.,* encrypting different nuggets with different cipher configurations in the same drive volume and even the same file. With Forward switching, we toggled nuggets between two cipher configurations to trade off battery life and security; with Selective switching, we want to select a specific cipher configuration each time we encrypt a nugget to trade off performance and security.

Table 1 summarizes what happens on I/Os when FLEXCRYPT is initialized using the Selective model with two available cipher configurations: the high-overhead high-security C_1 and the low-overhead reduced-security C_2. Suppose we have a multi-gigabyte corporate document that needs to be continually disseminated to and consumed by multiple parties. On writes, we want certain high priority secrets in this document to be available to only a subset of receivers, so we should use our powerful but resource-intensive C_1 to encrypt nuggets containing these secrets. On the other hand, the remainder of surrounding nuggets should use the more performant C_2, allowing other parties to interact with the low-priority data (perhaps on a lightweight mobile device) without experiencing degraded storage performance. On reads, FLEXCRYPT uses its own metadata to automatically determine which cipher configuration to use for decryption.

3.2 FLEXAPI: Cipher Wrapper Interfaces

One of the goals of FLEXCRYPT is that we might use any stream cipher regardless of its implementation details. However, allowing these ciphers to co-exist in the same volume is challenging since there are many cipher implementations that we might use, 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 [32]. Using OpenSSL's AES implementation in CTR mode requires manually tracking the counter state and individual ciphertext blocks are retrieved though corresponding function invocations [72]. 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 [12]. HC-128 and other ciphers have similarly disparate requirements.

Further, unlike prior work, FLEXCRYPT must 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, we must abstract away these input and output requirements by decoupling cipher implementations from the core encryption process. We present FLEXAPI, a collection of interfaces that allow implementors to write light (<100 LOC) wrapper functions around cipher implementations without modifications to third-party code; we call these wrapped ciphers *crypts*. Crypts present FLEXCRYPT with a uniform encryption and decryption interface across all cipher implementations, enabling normally incompatible ciphers to encrypt and decrypt arbitrary nuggets and thus coexist under FLEXCRYPT.

The ability for disparate cipher implementations to co-exist in this way forms the foundation for FLEXCRYPT's ability to switch the system between different cipher configurations efficiently and effectively. To facilitate this, FLEXAPI acts as a translation layer between the encryption module (where FLEXSWITCH is implemented) and storage (where FLEXAPI output is committed).

FLEXCRYPT receives I/Os from the operating system at the block device level like any other device-mapper. These I/Os come in the form of either reads or writes. When a read 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. When a write 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.

Crypts handle these I/Os by implementing either `xor_interface` or both `read_interface` and `write_interface`. These interfaces are implemented by crypt authors and later called by the encryption module when encrypting, re-ciphering, and decrypting nuggets during I/O.

`xor_interface` executes independently of FLEXCRYPT internals and treats encryption and decryption as the same operation. Crypts 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 the key buffer length will always be what the cipher implementation expects, alleviating the burden of key management. Similarly, the XOR buffer will be XOR-ed with the appropriate nugget contents automatically, alleviating the burden of drive access and other tedious calculations.

`read_interface` and `write_interface`, on the other hand, treat encryption and decryption as distinct concerns. `read_interface` handles decryption and re-ciphering during reads. `write_interface` handles encryption and re-ciphering during writes. Crypts receive full access to FLEXCRYPT internals, giving wrapper code deep hooks into the encryption and decryption process and allowing implementers to bypass parts of the nugget-based storage layout if necessary. This comes at the cost of increased code complexity and potential performance implications, since FLEXCRYPT must account for not having absolute control over its internal data structures when using this crypt.

For this work we have implemented 15 crypts using 7 ciphers—ChaCha, Rabbit, Sosemanuk, AES-CTR, and HC-128 with `xor_interface`; Freestyle and AES-XTS with `read_interface` and `write_interface`—each in under 100 LOC (excluding the cipher algorithm itself).

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*	0	1	0
Freestyle (F)	0	1.34	0
Freestyle (B)	0.5	1.67	0
Freestyle (S)	1	2	0

Table 2: **Evaluating Crypts.** 12 cipher crypts (§3.2) evaluated using the FLEXTRADE framework as described in §3.3.

3.3 FLEXTRADE: An Evaluation Framework

To reason about which crypt to use and when, non-expert users must have a way to evaluate and compare their utility in the novel context of heterogeneous FDE. However, different ciphers have a wide range of desirable security properties, performance profiles, and output characteristics, including those that randomize their outputs and those with non-length-preserving outputs. To simplify comparison, we propose FLEXTRADE, a novel evaluation framework that quantifies crypts according to three features relevant to generic heterogeneous FDE: *relative rounds*, which gives users a sense for the strength of the crypt; *ciphertext randomization*, which gives a sense for how safe a crypt's output is when encrypting "data in motion" (*e.g.,* to backup/snapshot; see §2); and *ciphertext expansion*, which gives users a sense for how a crypt will impact total available drive space. §3.4 describes how we might apply FLEXTRADE to reason about crypt choice.

Relative Rounds (Rounds). The cipher implementations we examine in this paper are all constructed around the notion of *rounds*, where a higher number of rounds (and/or larger key space) is positively correlated with a higher resistance to brute force given no fatal related-key or other attacks [61]; in other words: more rounds implies stronger encryption. Groups of crypts implement the same cipher but use different configurations, *e.g.,* the crypts for the ChaCha cipher configured for 8 rounds (ChaCha8), 12 rounds (ChaCha12), and the standard 20 rounds (ChaCha20). These crypts are scored across a discrete uniform distribution from 0, representing the lowest round configuration (*e.g.,* 8-round ChaCha) to 1, representing the highest round configuration (*e.g.,* 20-round ChaCha).

Table 2 shows the relative round score for twelve crypts grouped by the four ciphers they implement. ChaCha and Salsa each have three crypts that differ only by round count, yielding the scores *0, 0.5, and 1*. HC-X has two, yielding *0 and 1*. Freestyle has four crypts, but only three of them differ

by round count (*Freestyle (F)* and *Freestyle** use the same number of rounds). If a crypt is the singular implementation of some cipher, it receives a 1.

Ciphertext Randomization (Randomization). A cipher employing 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 confidentiality-violating schemes where the ciphertext is in full control of the adversary [12]. 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 §2). Ciphers without this property—such as ChaCha20 and AES-XTS—are trivially broken when key-nonce-plaintext 3-tuples are reused. In StrongBox, this is referred to as an "overwrite" [27].

Though there are many ways to achieve ciphertext randomization, the crypts 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 [12]. In configuring 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.

Table 2 shows the ciphertext randomization score for twelve crypts grouped by the four ciphers they implement. ChaCha, Salsa, and HC-X *do not employ ciphertext randomization at all*, so the crypts implementing them score a 0. Freestyle *does* employ ciphertext randomization (see §4), so the four crypts implementing them are scored from 1 (least randomized output) to 2 (most randomized output) across an even distribution similar to relative rounds yielding the scores *1, 1.34, 1.67, and 2*.

Ciphertext Expansion (Expansion). A cipher that exhibits ciphertext expansion is non-length-preserving: it outputs more bytes of ciphertext after encryption than were input. This can cause major problems in any FDE context. For instance, cryptosystems that are tightly coupled to AES-XTS (e.g. Linux's dm-crypt [8] and Microsoft's BitLocker [66]) or ChaCha (e.g. StrongBox [27], Google's Adiantum [23]) have storage layouts that hold length-preserving ciphertext 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. Further, since non-length-preserving ciphers *use more drive space* than length-preserving ciphers when encrypting the same plaintext, users will have less total writable drive space.

The crypts included in our analysis that exhibit ciphertext expansion have an overhead of around 1.56% per plaintext message block [12]. Hence, this is a binary feature in that a crypt either outputs ciphertext of the same length as its plaintext input or it does not. A crypt scores a 0 if it *is not* length-preserving or a 1 if it is, as shown in table 2.

3.4 Discussion

Trade score. With FLEXTRADE, we do not gain anything useful by comparing across dimensions; *i.e.,* it is unhelpful to compare one crypt's round score to another crypt's randomization score. However, we can consider the relative round count and ciphertext randomization scores together. We call this simple sum a crypt's *trade score*. This simple score gives users a quick and intuitive method to reason about which crypt to use and when: a lower trade score denote less rounds, less randomization, and higher performance while higher scores denote more rounds, more randomization, but lower performance.

Trade scores also have the useful feature where scores ≤ 1 can be regarded as not suitable for encrypting data in motion—a property that is necessary for safe encrypted backups like in the battery saver mode example (see §2).

Secure metadata management. The focus of this work is to implement mechanisms and policies to perform flexible switching of cipher configurations. These configurations are built atop an existing block-level encryption module that manages our data structures and provides cryptographic support. There were several open source choices for an encryption module: Linux's encryption device mapper [1, 8], encryption-ready F2FS filesystem [50], or StrongBox [27]. We decided to build FLEXCRYPT atop StrongBox because it already implements stream cipher based FDE and employs a nugget-based drive layout.

FLEXCRYPT depends on the data structure management provided by StrongBox, such as its *transaction* and *rekeying journals* to avoid overwrite violations [27], *Merkle tree* to avoid rollback and related attacks [27], *monotonic counter* on a trusted hardware to prevent rollbacks, *keycount store* to derive unique encryption keys, and *per-nugget metadata* to store cipher-specific metadata (useful for ciphertext-expanding ciphers). Every FLEXCRYPT volume also has a "head" area that stores implementation-wide data like the master encryption key or which cipher is currently active.

While we reuse some StrongBox components, all of these are tightly integrated to with a special ChaCha20 implementation. We had to untangle this, hence the FLEXAPI contribution, where we expose structured interfaces allowing cipher implementors to easily build crypts from any off-the-shelf stream cipher implementation. Specifically, of the six StrongBox components listed above, only the keycount store and monotonic counter can be reused as is; the others were rewritten to support FLEXSWITCH and FLEXAPI.

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 work, encryption is achieved via a binary additive approach: cipher output (keystream) is combined with plaintext nugget contents using XOR, generating confidential outputs, with metadata to track writes and ensure 1) that pad reuse never occurs and 2) that the system can recover from crashes into a secure state. Hence, confidentiality is guaranteed.

In terms of *integrity*, an adversary should not be able to tamper with drive state and it go unnoticed. Nugget integrity is guaranteed by StrongBox's in-memory Merkle tree [27].

These switching models add an additional security concern: even if we switch the drive to a new crypt, depending on the switching model, there may still be data on the drive that was encrypted with an old crypt. Does this create a confidentiality problem? For the Forward model, this implies any nugget may at any time be encrypted using the old crypt. For the Mirrored model, the volume is divided into regions where nuggets use a specific crypt per region; hence, there exists at least one region using the old crypt. However, like StrongBox, since nuggets are encrypted and decrypted independently, the confidentiality guarantee of FLEXCRYPT can be reduced to the confidentiality guarantee of the "weakest" cipher [27]. Hence, there is no problem unless the user purposely chooses a broken cipher. For the Selective model, there is never a problem since the user directly selects which crypt encrypts which nugget every time.

Higher-level integration. FLEXCRYPT expects a higher-level integration/policy that communicates what kind of switching should be performed and when. An example is the battery saver scenario, where the OS battery saver application is expected to notify FLEXCRYPT when to move to an energy-efficient crypt versus a backup-ready crypt using Forward switching. See §6 for more integrations.

Generality. FLEXCRYPT can be seen as a drop-in replacement for the popular Linux dm-crypt layer. For performance reasons, like StrongBox, FLEXCRYPT recommends a Log-structured File System (LFS) such as F2FS, YAFFS, or LogFS, 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 [27]. Though FLEXCRYPT is a software solution, the same LFS-based logic can be adopted by hardware like flash devices. For example, the LFS-like "no in-place updates" nature of Flash Transition Layers (FTL) would allow heterogeneous FDE implementations be performant at the device controller level, allowing the use of 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 [1]. We deploy FLEXCRYPT on top of the BUSE virtual block device [7] as our 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.

Cipher	Source
AES-XTS, AES-CTR	OpenSSL [72] & libsodium [26]
ARM Neon ChaCha	Floodyberry [32]
Freestyle	Babu [12]
eSTREAM Profile 1	libestream [58]

Table 3: **Cipher Implementations.** See §4, §3.2.

Among the many ciphers FLEXCRYPT supports and that we implemented as crypts over the course of this research, we focus on the five crypts described here for our evaluation and cases studies: ChaCha8, ChaCha20 [15], and Freestyle [12] 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). Note that we also implemented a "vacuous" mode using minimum parameters (*Freestyle** in table 2), but this mode is not included in our evaluation.

Table 3 shows the cipher implementations we use wrapped into crypts with FLEXAPI; see §3.2.

5 Evaluation

In this section we answer the following questions:

1. **Why switch?** What shape is the baseline security-latency crypt tradeoff space? (§5.1)
2. **Does heterogeneous FDE work?** Can FLEXCRYPT support performant flexible cipher switching? (§5.2)
3. **At what cost?** What is the overhead of switching? (§5.3)

Experimental Setup. We implement FLEXCRYPT 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 crypts.

Figure 1: **Baseline I/O performance.** Median sequential and random I/O baseline as discussed in §5.1.

Figure 2: **Forward I/O performance.** Median sequential and random I/O compared to baseline. Each cluster of 3 dots between configurations represents the 7:3, 5:5, and 3:7 ratios as discussed in **??**.

Figure 3: **Mirrored and Selective I/O performance.** Median sequential and random I/O compared to baseline. Each cluster of 3 dots between configurations represents the 7:3, 5:5, and 3:7 ratios as discussed in **??**.

Methodology. 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 crypt (720 runs per *ratio*, see below), 30 results per byte size, 120 results per workload. Results are accumulated and the median is taken. All experiments are performed with basic Linux I/O commands, bypassing system caching.

When evaluating switching performance, a finer breakdown in workloads is made using a pre-selected pair of crypts we call the *primary* and *secondary*. FLEXCRYPT is initialized using the primary crypt. Once we trigger a switch, we switch to the secondary crypt. The switch is triggered according to a certain *ratio* of I/O operations. For example: given 10 40MB "write-read" (write and then read back) operations, we may write-read 40MB 3 times using the primary crypt, initiate a switch, and then write-read 40MB 7 times. This would be a 3:7 ratio. There are three ratios we use to evaluate switching performance: 7:3, 5:5, and 3:7.

5.1 Baseline Crypt Performance

Figure 1 shows the baseline tradeoff between trade score (x-axis; see also §3.4) and median normalized latency (y-axis) of different crypts 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 crypts: ChaCha8 (C8), ChaCha20 (C20), Freestyle Fast (FF), Freestyle Balanced (FB), and Freestyle Strong (FS). Of these crypts, those with higher trade scores resulted in higher overall latency and increased energy use for I/O operations. The relationship between these concerns is not linear across crypts, 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 here. Due to the overhead of metadata management and the fast completion time of the 4KB workloads (*i.e.,* little time for amortization of overhead), C8 and C20 take longer to complete than FF. This advantage is not enough to make FB or Secure workloads complete faster than the ChaCha variants, however.

Though C8 is faster than C20, there is some variability in our timing setup when capturing extremely fast events occurring close together in time. This is why C8 sometimes appears with higher latency than C20 for normalized 4KB workloads. C8 is not slower than C20.

5.2 Heterogeneous FDE Performance

Figure 2 shows the tradeoff between trade score (x-axis; see also §3.4) and median normalized latency (y-axis) of different crypts for our sequential and random I/O workloads using Forward switching. After a certain number of write-read operations using the primary crypt, a switch is initiated and FLEXCRYPT begins using the secondary crypt for I/O. For each of the four pairs of crypts in the figure (primary+secondary: C8+C20, C20+FF, FF+FB, FB+FS), this is repeated three times: once at every ratio point *between* our baseline crypt measurements (*i.e.,* 7:3, 5:5, and 3:7 described above). Ratio points above baseline measure overhead.

The point of this experiment is to determine if FLEXCRYPT can switch the encryption efficiently enough to support heterogeneous FDE—that is, contiguous nuggets encrypted with different crypts—without devastating performance. For the 40MB, 5MB, and 512KB workloads (40MB is shown), we see that overhead is low, demonstrating that with heterogeneous FDE we can in fact achieve flexible security/energy tradeoffs unachievable with prior work and all with minimal overhead.

Again, due to the overhead of metadata management and fast completion time of the workloads, C8 and C20 take longer to complete than FF for 4KB reads. We also see very high

latency for ratios between FF and FS crypts. This is because Freestyle is non-length-preserving, so extra operations must be performed on every write, and the algorithm itself is generally much slower than the ChaCha variants.

Figure 3 shows the performance of the Mirrored and Selective models vs baseline with the same configuration of ratios as fig. 2.

For the 40MB, 5MB, and 512KB workloads (40MB is shown), we see that Mirrored and Selective *read* workloads and the Selective *write* workload have similar latency overhead to the Forward model experiments. This makes sense, as most of the overhead for Selective and Mirrored reads is determining which part of the drive to commit data to. The same applies to Selective writes. For the 4KB Mirrored and Selective *read* workloads and the Selective *write* workload, we see behavior similar to that in fig. 2, as expected.

Mirrored writes across all workloads are slow. This is to be expected since writes are being duplicated. This overhead is highest for the 4KB Mirrored write workload.

5.3 Summary Discussion

We calculate that Forward switching has average latency overhead at 0.08x/0.10x (read/write) for 40MB, 5MB and 512KB workloads compared to baseline I/O, demonstrating FLEXCRYPT's amortization of switching costs. Average overhead is 0.38x/0.44x (read/write) for 4KB workloads, where FLEXCRYPT is unable to amortize cost. Similarly, we calculate that Selective switching has average overhead at [TODO: 0x/Zx] (read/write) for 40MB, 5MB and 512KB workloads compared to baseline I/O. Average overhead is [TODO: Yx/Zx] (read/write) for 4KB workloads. Finally, we calculate that Mirrored switching has average overhead at [TODO: Yx/Zx] (read/write) for 40MB, 5MB and 512KB workloads compared to baseline I/O, with high write latency due to mirroring. Average overhead is [TODO: Yx/Zx] (read/write) for 4KB workloads.

In summary, these results show that FLEXCRYPT enables a wide range of performance and security tradeoffs unachievable with prior work. This useful feature comes at a cost of at most [TODO: Yx/Zx] increased read/write overhead (excluding Mirrored) compared to existing FDE approaches like StrongBox, and at the cost of reduced performance for very small workloads. We conclude that any FDE workload dominated by I/Os larger than 4KB would benefit from the increased flexibility of FLEXCRYPT and heterogeneous FDE.

6 Case Studies

In this section we conclude our case studies. See §5 for the setup and methodology behind each study.

Figure 4: **Battery saver mode.** Energy-security tradeoff given strict energy budget as discussed in §6.1.

6.1 "Battery Saver" Mode

In this first case study, we revisit the motivational Forward switching example (§2). Our goal is to complete a workload within a strict energy budget. Here, our netbook's storage will use *Freestyle Balanced* when operating normally and *ChaCha8* when battery saver mode is activated.

To simulate this, we repeat the following twice: 1) we begin writing 10 40MB files using Freestyle Balanced. 2) After five seconds, our device simulates battery saver mode by pinning FLEXCRYPT processes to the energy-efficient LITTLE cores and underclocking them to lowest frequency. 3) We complete the workload.

On the first run, the workload is completed normally. On the second run, when entering battery saver mode, we additionally trigger a switch to the ChaCha8 crypt using the Forward switching model.

Figure 4 shows the total energy used after completing both runs. *FB only* (left) favors security even when backups are paused while *Forward (FB+C8)* (right) performs the switch when the system enters battery saver mode. These results show that 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.

6.2 No-Downtime Encryption Upgrade

In this second case study we revisit the Mirrored switching example (§3.1). Our goal is to upgrade live storage from one cipher to another without downtime or degraded service. Our operator wants to upgrade from *ChaCha20* to *Freestyle Fast*.

To simulate pre- and post-migration state, we execute 10 5MB write-read operations to two FLEXCRYPT instances using ChaCha20 and Freestyle Fast respectively. To simulate the activity during migration, we repeat the operations on a third instance using Mirrored switching.

Figure 5 shows the overhead of Mirrored switching on our system's performance. On the left we see similar read latency between pre-, during-, and post-migration workloads of [TODO: X] s. On the right, we see write latency increase by [TODO: X] s during the migration but recover post-migration. Thus, Mirrored switching allows us to transition our storage from one cipher to another without interruption or egregious overhead.

Figure 5: **No-downtime encryption upgrade.** Upgrading storage system encryption with zero downtime as discussed in §6.2.

Figure 6: **Scalable encryption.** Filesystem-agnostic file-level encryption as discussed in §6.3.

6.3 Scalable Encryption

In this final case study we revisit the Selective switching example (§3.1). Our goal is to achieve a performance win by ensuring that the only data encrypted using the strongest highest-overhead crypt is the small percentage of data that needs it. We note the Selective switching model requires users to annotate certain files with a special tag via file system calls and 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 crypt.

We begin with 10 5MB and 10 4KB write-read operations to two FLEXCRYPT instances: one using *ChaCha8*, a high performance crypt, and the other using *Freestyle Strong*, a high security crypt. We repeat the operations on a third instance using Selective switching where 30% of the data is considered highly sensitive and uses Freestyle Strong.

C8 only (leftmost) and *FS only* (rightmost) in fig. 6 show the two extremes: ChaCha8 exhibits low latency, consistently less than 1s across workloads, while Freestyle Strong exhibits 5s to 20s latency depending on the workload. *Selective (C8+FS)* (middle) shows a reduction of 3.1x to 4.8x for read latency and 1.6x to 2.8x for write latency, all without compromising on security. Thus, Selective switching keeps our sensitive data at the mandated security level while keeping the performance benefits of using a fast crypt.

7 Related Work

Stream cipher based FDE. The standard approach to FDE, using AES-XTS, introduces significant overhead [25]. Recently, it has been established that encryption using *stream ciphers* for FDE is faster than using AES [27], but accomplishing this in practice is non-trivial. Prior work explores several approaches: a non-deterministic CTR mode (Freestyle [12]), a length-preserving "tweakable super-pseudorandom permutation" (Adiantum [23]), and a stream cipher in a binary additive (XOR) mode leveraging LFS overwrite-averse behavior to prevent overwrites (StrongBox [27]).

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 models and crypts; instead of myopically pursuing a performance win, FLEXCRYPT gives us the flexibility to pursue energy/battery and security wins as well.

Trading security for other concerns. Further, trading off security for energy, performance, and other concerns is not a new research area [34, 37, 53, 64, 86, 88, 89]. Goodman et al. introduced selectively decreasing the security of some data to save energy [34]. However, their approach is designed for communication and only considered iteration/round count, thus it did not anticipate the need for FLEXSWITCH, FLEXAPI, or FLEXTRADE. Wolter and Reinecke study approaches to quantifying security in several contexts [86]. 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 [23]. While not an FDE solution, LastPass has dealt with scaling the number of iterations of PBKDF#2, trading security for performance during login sessions [44].

Energy-aware mobile devices. On mobile energy/battery life, prior works have shown the importance of energy-saving mode and bugs that drain energy [59, 92]. Additional work shows that significant energy savings can be achieved by dynamically adapting resource usage to the current workload [31,42,48,68]. That work, however, is almost entirely focused on the compute and memory subsystems. FLEXCRYPT brings the benefits of a dynamic energy-saving mode to the storage layer.

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 heterogeneous FDE, we 1) identified three switching modes to switch ciphers with low overhead and no downtime, 2) demonstrated an interface that allows multiple stream ciphers to be composed in a generic manner, and 3) proposed a quantification framework for determining when to use one cipher over another. Our case studies show how different switching modes can be used to optimize for different goals. We believe heterogeneous encryption will become increasingly important as users demand storage systems be able to balance

conflicting operational requirements. We hope this work inspires further research in achieving this balance. Our work is publicly available open source [1].

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] EMBEDDED MULTI-MEDIA CARD (e•MMC), ELECTRICAL STANDARD (5.1), 2015.

[10] 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.

[11] Apple. Use low power mode to save battery life on your iphone, December 2019.

[12] 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.

[13] 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.

[14] 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.

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

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

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

[18] 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.

[19] 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.

[20] 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.

[21] Cloudera. Upgrading cloudera navigator encrypt.

[22] 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.

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

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

[25] Joan Daemen and Vincent Rijmen. AES proposal: Rijndael. 1999.

[26] Frank Denis. Libsodium 1.0.12.

[27] 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.

[28] 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.

[29] 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.

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

[31] Jason Flinn and M. Satyanarayanan. Energy-aware adaptation for mobile applications. In *Proceedings of the Seventeenth ACM Symposium on Operating Systems Principles*, SOSP '99, page 4863, New York, NY, USA, 1999. Association for Computing Machinery.

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

[33] 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.

[34] 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.

[35] Google. Get the most life from your android device's battery - android help.

[36] Google. Use low power mode to save battery life on your iphone, December 2019.

[37] 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.

[38] 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.

[39] 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.

[40] HCL.

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

[42] Henry Hoffmann. Jouleguard: Energy guarantees for approximate applications. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, page 198214, New York, NY, USA, 2015. Association for Computing Machinery.

[43] Huawei. How do I upgrade the AES encryption algorithm to AES256.

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

[45] 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.

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

[47] Ryusuke Konishi, Yoshiji Amagai, Koji Sato, Hisashi Hifumi, Seiji Kihara, and Satoshi Moriai. The linux implementation of a log-structured file system. *SIGOPS Oper. Syst. Rev.*, 40(3):102–107, July 2006.

[48] Etienne Le Sueur and Gernot Heiser. Slow down or sleep, that is the question. In *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'11, page 16, USA, 2011. USENIX Association.

[49] 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.

[50] Changman Lee, Dongho Sim, Jooyoung Hwang, and Sangyeun Cho. F2FS: A new file system for flash storage. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 273–286, Santa Clara, CA, 2015. USENIX Association.

[51] LG. Battery saver mode.

[52] 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.

[53] 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.

[54] 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.

[55] ARM Limited. ARM security technology: Building a secure system using trustzone technology, 2009. PRD29-GENC-009492C.

[56] 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.

[57] 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.

[58] Lvella. lvella/libestream.

[59] Xiao Ma, Peng Huang, Xinxin Jin, Pei Wang, Soyeon Park, Dongcai Shen, Yuanyuan Zhou, Lawrence K. Saul, and Geoffrey M. Voelker. edoctor: Automatically diagnosing abnormal battery drain issues on smartphones. In Nick Feamster and Jeffrey C. Mogul, editors, *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2013, Lombard, IL, USA, April 2-5, 2013*, pages 57–70. USENIX Association, 2013.

[60] 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.

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

[62] 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.

[63] 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.

[64] 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.

[65] Microsoft. Windows hardware dev center: Battery saver.

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

[67] 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.

[68] Nikita Mishra, Connor Imes, John D. Lafferty, and Henry Hoffmann. Caloree: Learning control for predictable latency and low energy. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '18, page 184198, New York, NY, USA, 2018. Association for Computing Machinery.

[69] 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.

[70] 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.

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

[72] OpenSSL. OpenSSL 1.1.0h.

[73] 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.

[74] 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.

[75] 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.

[76] 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.

[77] 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.

[78] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. *ACM Trans. Comput. Syst.*, 10(1):26–52, February 1992.

[79] Adam Ruprecht, Danny Jones, Dmitry Shiraev, Greg Harmon, Maya Spivak, Michael Krebs, Miche Baker-Harvey, and Tyler Sanderson. VM live migration at scale. In *Proceedings of the 14th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '18, pages 45–56, New York, NY, USA, 2018. Association for Computing Machinery.

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

[81] 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.

[82] 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.

[83] Global Platform Device Technology. TEE client API specification version 1.0, 2010. GPD_SPE_007.

[84] 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.

[85] 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.

[86] 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.

[87] 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.

[88] 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.

[89] 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.

[90] 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.

[91] 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.

[92] Qingbo Zhu, Francis M. David, Christo Frank Devaraj, Zhenmin Li, Yuanyuan Zhou, and Pei Cao. Reducing energy consumption of disk storage using power-aware cache management. In *10th International Conference on High-Performance Computer Architecture (HPCA-10 2004), 14-18 February 2004, Madrid, Spain*, pages 118–129. IEEE Computer Society, 2004.

[93] 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.