US 20190238312A1

(19) United States
(12) Patent Application Publication
Dickens, III et al.
(10) Pub. No.: US 2019/0238312 A1
(43) Pub. Date: Aug. 1, 2019

(54) **STREAM CIPHERS FOR DIGITAL STORAGE ENCRYPTION**

(71) Applicant: **The University of Chicago**, Chicago, IL (US)

(72) Inventors: **Bernard Dickens, III**, Chicago, IL (US); **Haryadi Gunawi**, Chicago, IL (US); **Ariel Feldman**, Chicago, IL (US); **Henry Hoffmann**, Chicago, IL (US)

(21) Appl. No.: **16/264,991**

(22) Filed: **Feb. 1, 2019**

Related U.S. Application Data

(60) Provisional application No. 62/625,148, filed on Feb. 1, 2018.

Publication Classification

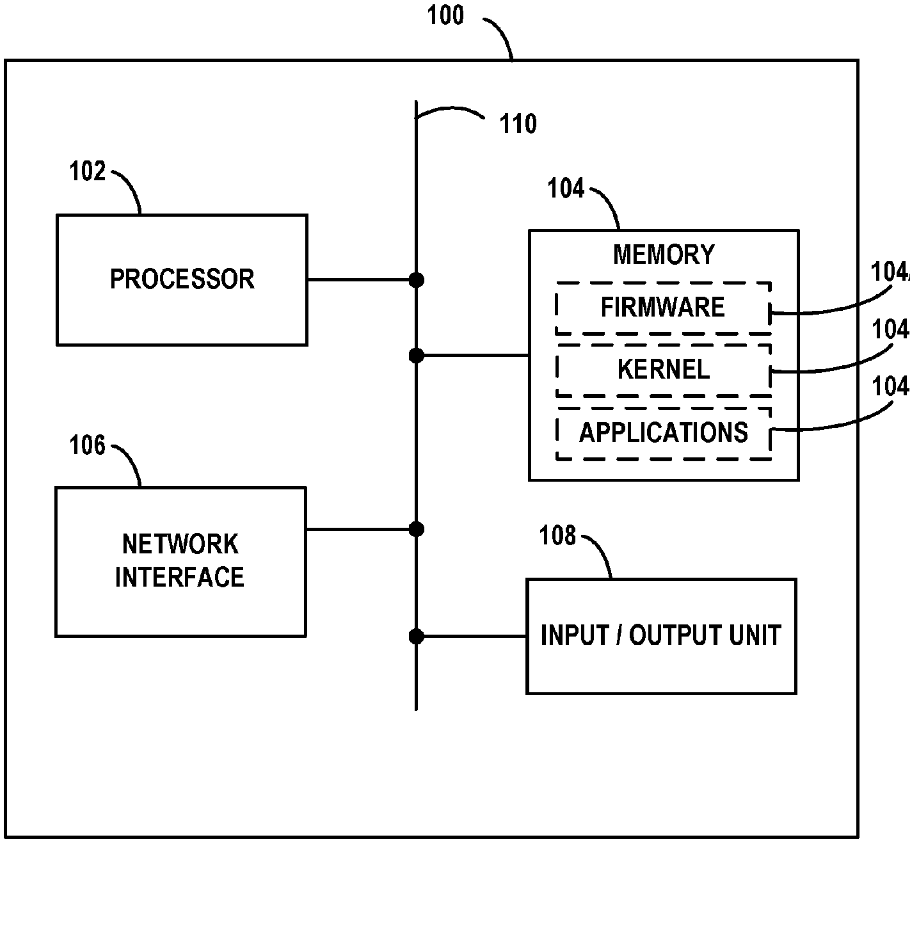
(51) **Int. Cl.**

H04L 9/06	(2006.01)
H04L 9/08	(2006.01)
G06F 21/60	(2006.01)
G06F 16/18	(2006.01)
G06F 3/06	(2006.01)

(52) **U.S. Cl.**
CPC ***H04L 9/065*** (2013.01); ***H04L 9/0861*** (2013.01); ***G06F 21/602*** (2013.01); ***H04L 9/0643*** (2013.01); ***G06F 3/0673*** (2013.01); ***G06F 16/1815*** (2019.01); ***G06F 3/0604*** (2013.01); ***G06F 3/064*** (2013.01); ***G06F 3/0659*** (2013.01); ***H04L 9/0618*** (2013.01)

(57) **ABSTRACT**

An embodiment involves receiving a request to write data to a memory unit. The memory unit is divided into one or more logical blocks, each subdivided into groups of sub-blocks encrypted in accordance with a stream cipher. The memory unit maintains a transaction journal that marks each sub-block as dirty or clean. The memory unit stores keycount values for each of the logical blocks. The embodiment also involves: determining that the request seeks to write a portion of the data to a particular sub-block marked as dirty in the transaction journal, decrypting the particular logical block in accordance with the stream cipher, writing the portion of the data to the particular sub-block, incrementing the keycount value of the particular logical block, encrypting the particular logical block using the stream cipher, a key, and the keycount value, and writing the particular logical block to the memory unit.



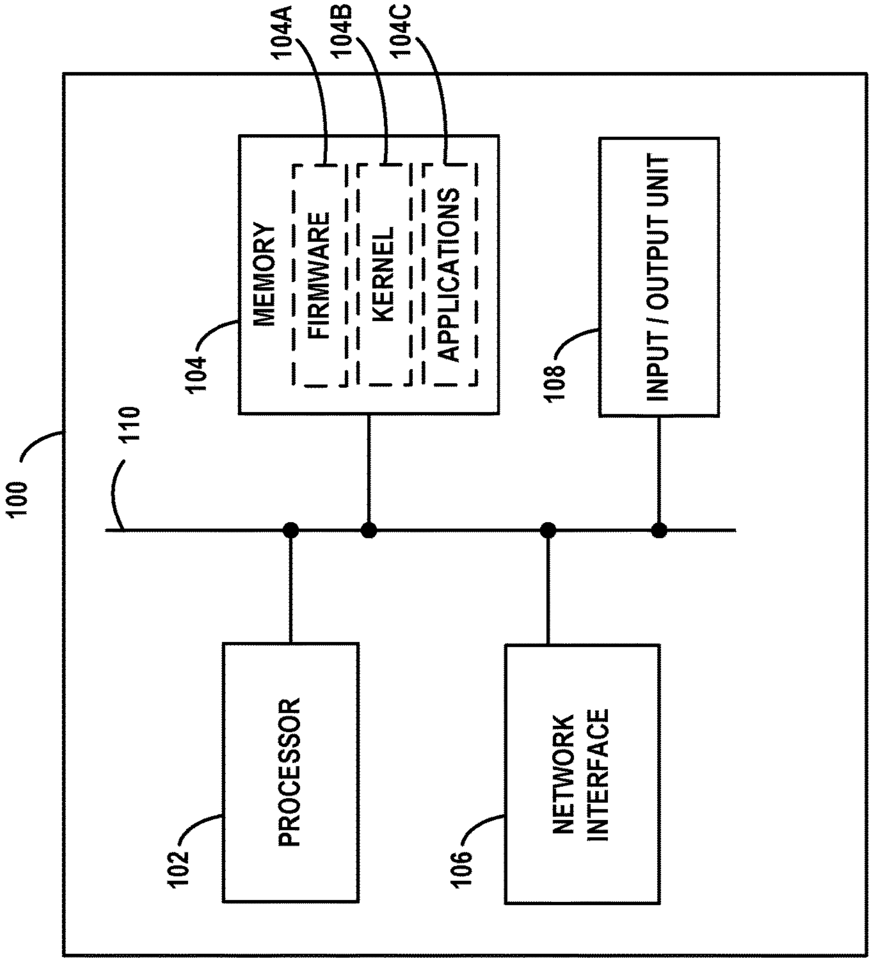

100
110
104
MEMORY
FIRMWARE
KERNEL
APPLICATIONS
104A
104B
104C
108
INPUT / OUTPUT UNIT
102
PROCESSOR
106
NETWORK INTERFACE

FIG. 1

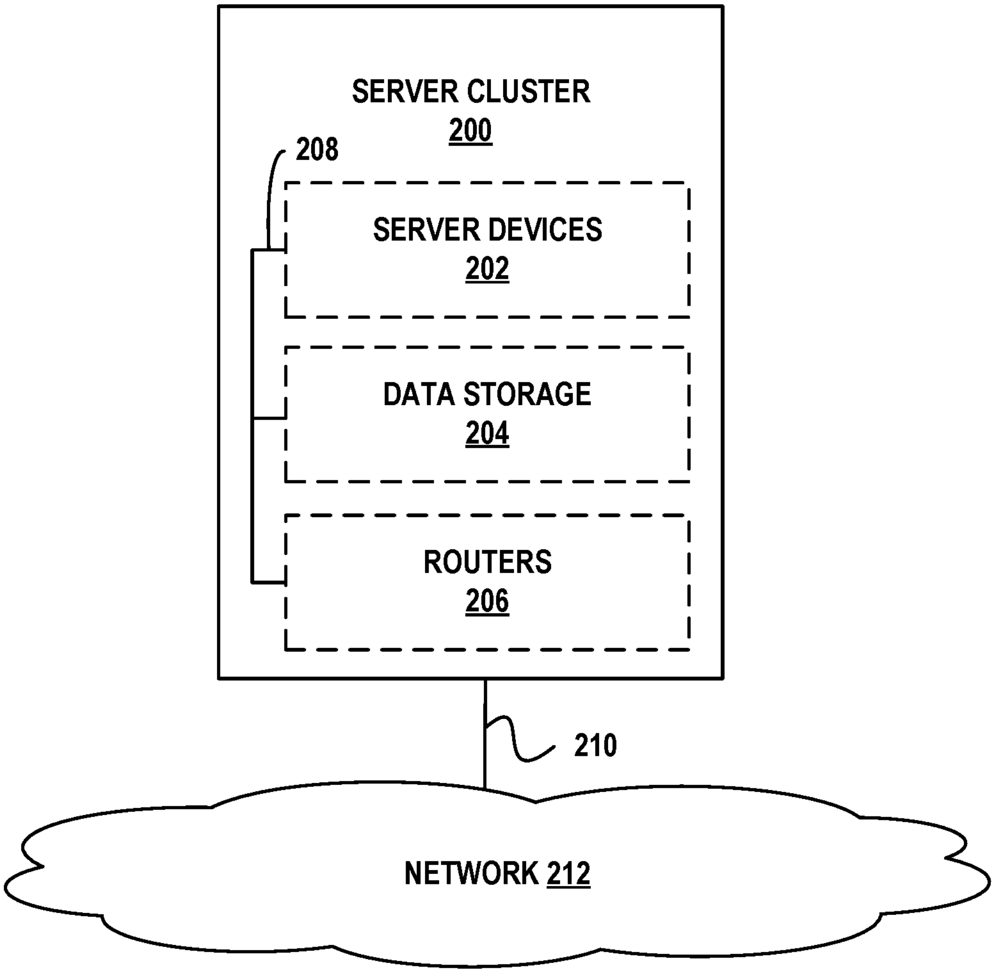

SERVER CLUSTER
200
208
SERVER DEVICES
202
DATA STORAGE
204
ROUTERS
206
210
NETWORK 212

FIG. 2

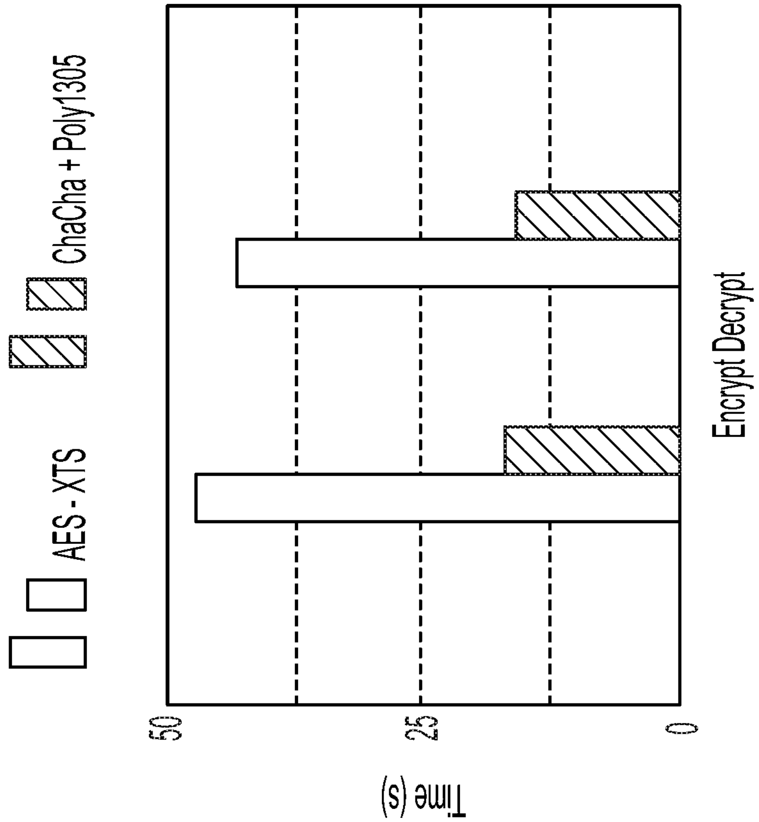


FIG. 3

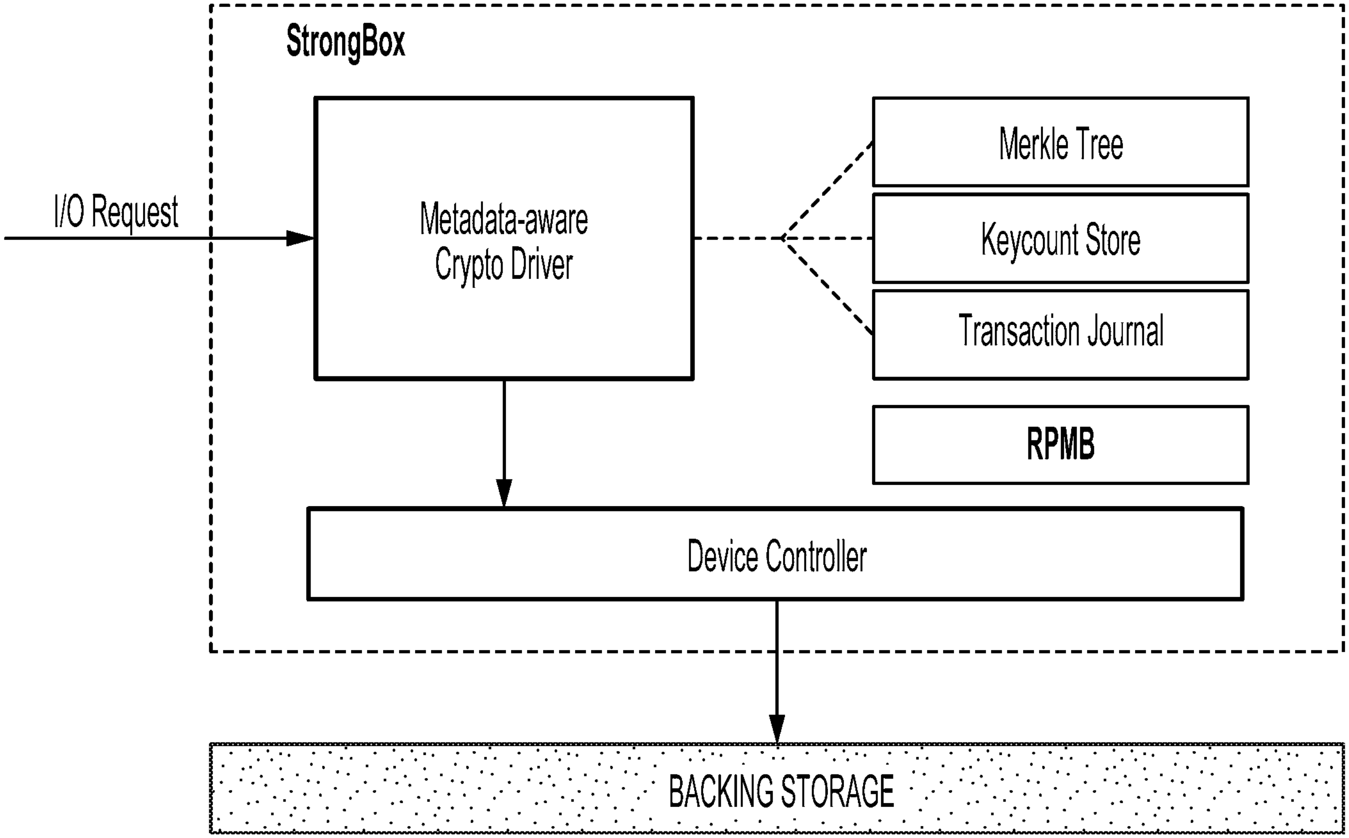

StrongBox
I/O Request
Metadata-aware Crypto Driver
Merkle Tree
Keycount Store
Transaction Journal
RPMB
Device Controller
BACKING STORAGE

FIG. 4

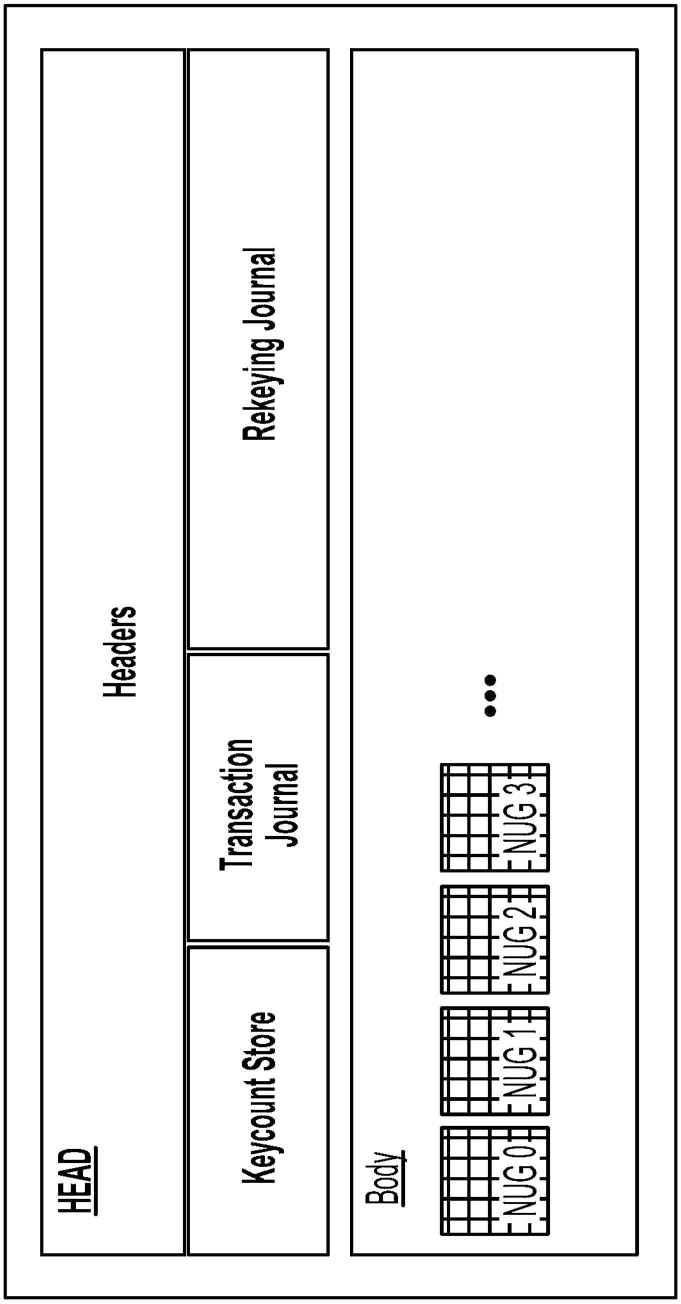
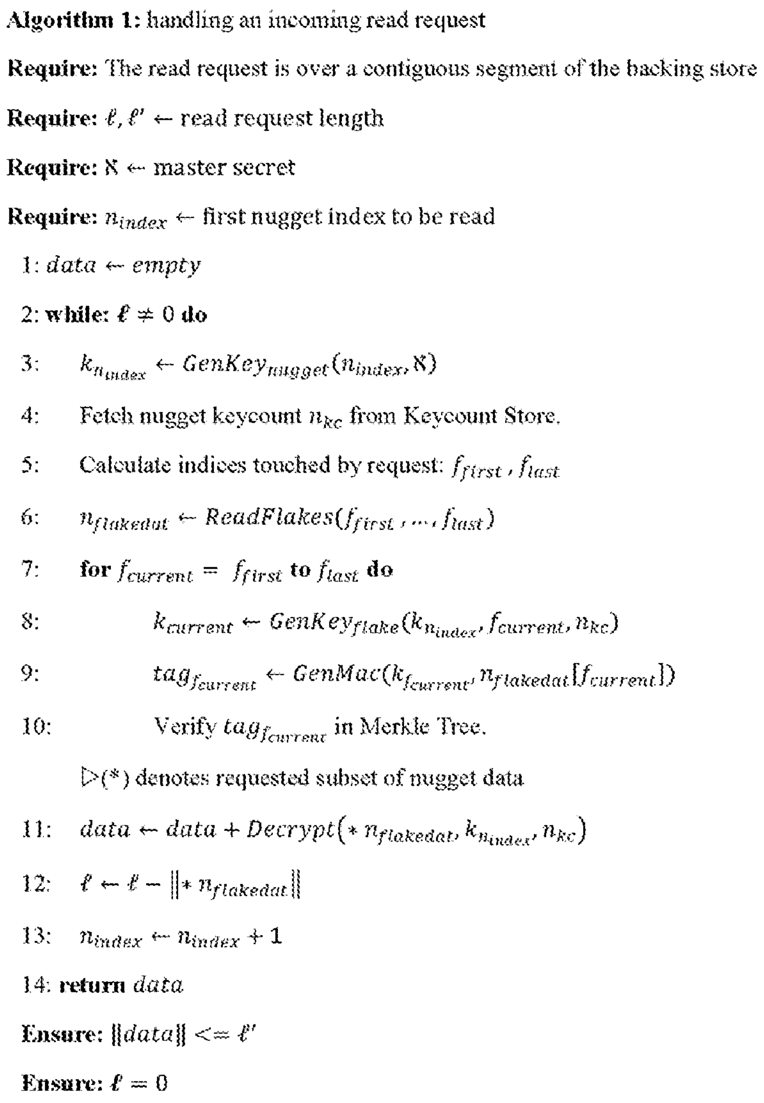

HEAD
Headers
Keycount Store
Transaction Journal
Rekeying Journal
Body
NUG 0
NUG 1
NUG 2
NUG 3
•••

FIG. 5



```
Algorithm 1: handling an incoming read request
Require: The read request is over a contiguous segment of the backing store
Require: $\ell, \ell' \leftarrow$ read request length
Require: $\aleph \leftarrow$ master secret
Require: $n_{index} \leftarrow$ first nugget index to be read
$data \leftarrow empty$
while: $\ell \neq 0$ do
    $k_{n_{index}} \leftarrow GenKey_{nugget}(n_{index}, \aleph)$
    Fetch nugget keycount $n_{kc}$ from Keycount Store.
    Calculate indices touched by request: $f_{first}, f_{last}$
    $n_{flakedat} \leftarrow ReadFlakes(f_{first}, \cdots, f_{last})$
    for $f_{current} = f_{first}$ to $f_{last}$ do
        $k_{current} \leftarrow GenKey_{flake}(k_{n_{index}}, f_{current}, n_{kc})$
        $tag_{f_{current}} \leftarrow GenMac(k_{f_{current}}, n_{flakedat}[f_{current}])$
        Verify $tag_{f_{current}}$ in Merkle Tree.
    ▷(*) denotes requested subset of nugget data
    $data \leftarrow data + Decrypt(* n_{flakedat}, k_{n_{index}}, n_{kc})$
    $\ell \leftarrow \ell - \|* n_{flakedat}\|$
    $n_{index} \leftarrow n_{index} + 1$
return $data$
Ensure: $\|data\| <= \ell'$
Ensure: $\ell = 0$
```

FIG. 6A

```
**Algorithm 2:** handling an incoming write request
**Require:** The write request is to a contiguous segment of the backing store
**Require:** $\ell, \ell' \leftarrow$ write requested length
**Require:** $\aleph \leftarrow$ master secret
**Require:** $data \leftarrow$ cleartext data to be written
**Require:** $n_{index} \leftarrow$ first nugget index to be affected
Increment secure counter: by 2 if recovering from a crash, else 1
**while:** $\ell \neq 0$ **do**
    Calculate indices touched by request: $f_{first}, f_{last}$
    **if** Transaction Journal entries for $f_{first}, \ldots, f_{last} \neq 0$ **then**
        Trigger rekeying procedure (see: Algorithm 3).
        **continue**
    Set Transaction Journal entries for $f_{first}, \ldots, f_{last}$ to 1
    $k_{n_{index}} \leftarrow GenKey_{nugget}(n_{index}, \aleph)$
    Fetch nugget keycount $n_{kc}$ from Keycount Store.
    **for** $f_{current} = f_{first}$ **to** $f_{last}$ **do**
        $n_{flakedat} \leftarrow empty$
        **if** $f_{current} == f_{first} \,||\, f_{current} == f_{last}$ **then**
            $n_{flakedat} \leftarrow CryptedRead(FSIZE, \aleph, n_{index}@f_{offset})$
        $n_{flakedat} \leftarrow Encrypt(n_{flakedat}, k_{n_{index}}, n_{kc})$
        $k_{f_{current}} \leftarrow GenKey_{flake}(k_{n_{index}}, f_{current}, n_{kc})$
        $tag_{f_{current}} \leftarrow GenMac(k_{f_{current}}, n_{flakedat})$
        Update new $tag_{f_{current}}$ in Merkle Tree.
        $WriteFlake(f_{current}, n_{flakedat})$
▷(*) denotes requested subset of nugget data if applicable
        $\ell \leftarrow \ell - \|{*}\, n_{flakedat}\|$
    $n_{index} \leftarrow n_{index} + 1$
Update and commit metadata and headers
**Ensure:** $\ell = 0$
```

FIG. 6B

```
**Algorithm 3:** rekeying process.
**Require:** The original write applied to a contiguous backing store segment
**Require:** $\ell \leftarrow$ write requested length
**Require:** $\aleph \leftarrow$ master secret
**Require:** $data \leftarrow$ cleartext data to be written
**Require:** $n_{index} \leftarrow$ nugget rekeying target
                                        ▷Read in and decrypt the entire nugget
$n_{nuggetdat} \leftarrow CryptedRead(NSIZE, \aleph, n_{index})$
Calculate indices touched by request: $f_{first}, f_{last}$
Write $data$ into $n_{nuggetdat}$ at proper offset with length $\ell$
Set Transaction Journal entries for $f_{first}, \ldots, f_{last}$ to 1
$k_{n_{index}} \leftarrow GenKey_{nugget}(n_{index}, \aleph)$
Fetch nugget keycount $n_{kc}$ from Keycount Store. Increment it by one.
$n_{nuggetdat} \leftarrow Encrypt(n_{nuggetdat}, k_{n_{index}}, n_{kc})$
Commit $n_{nuggetdat}$ to the backing store
                                        ▷Iterate over all flakes in the nugget
**for all** flakes $f_{current}$ **in** $n_{index}$ **do**
    $k_{f_{current}} \leftarrow GenKey_{flake}(k_{n_{index}}, f_{current}, n_{kc})$
    Copy $f_{current}$ data from $n_{nuggetdat} \rightarrow n_{flakedat}$
    $tag_{f_{current}} \leftarrow GenMac(k_{f_{current}}, n_{flakedat})$
    Update new $tag_{f_{current}}$ in Merkle Tree.
Update and commit metadata and headers.
```

FIG. 6C

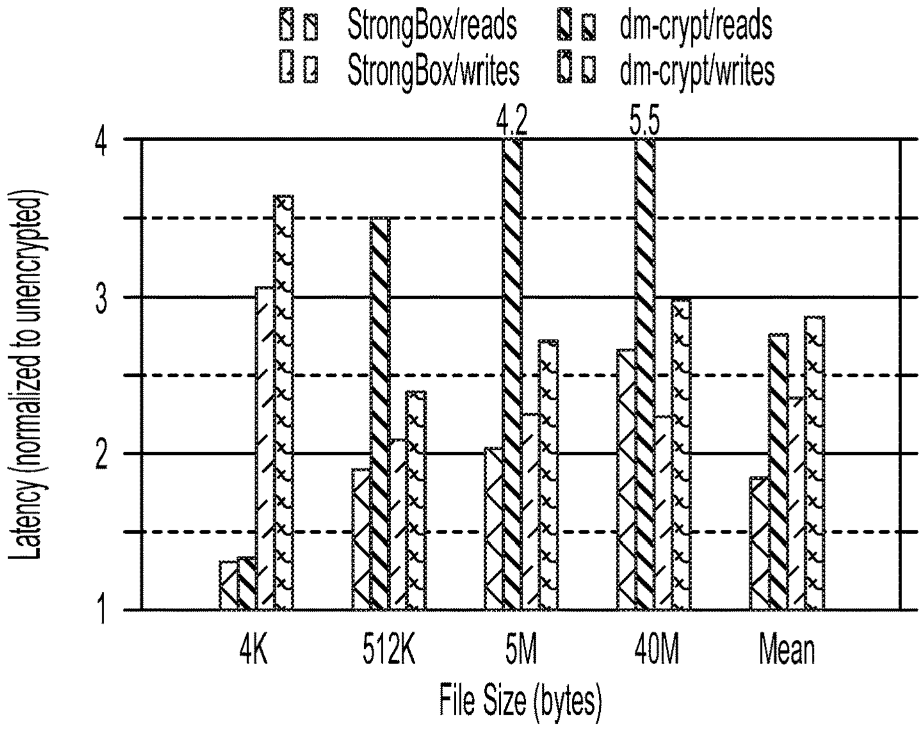

StrongBox/reads
dm-crypt/reads
StrongBox/writes
dm-crypt/writes
4.2
5.5
4
3
2
1
Latency (normalized to unencrypted)
4K
512K
5M
40M
Mean
File Size (bytes)

FIG. 7A

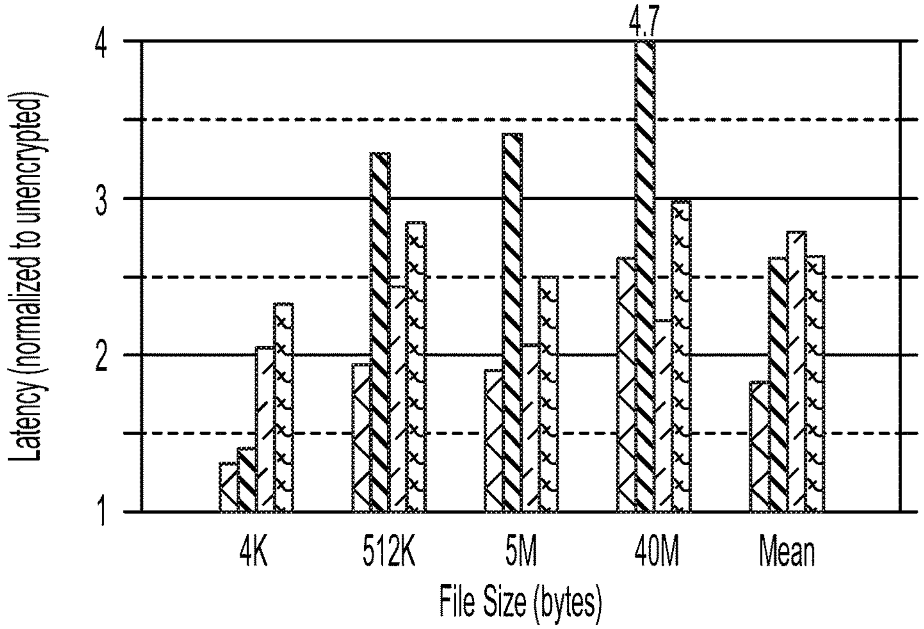

4.7
4
3
2
1
Latency (normalized to unencrypted)
4K
512K
5M
40M
Mean
File Size (bytes)

FIG. 7B

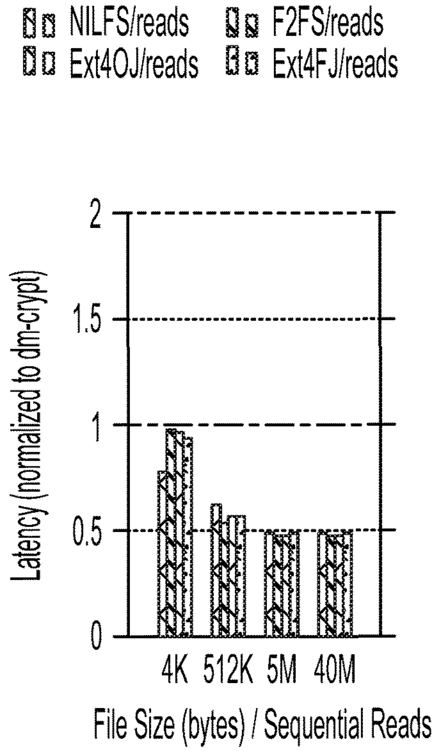


FIG. 8A

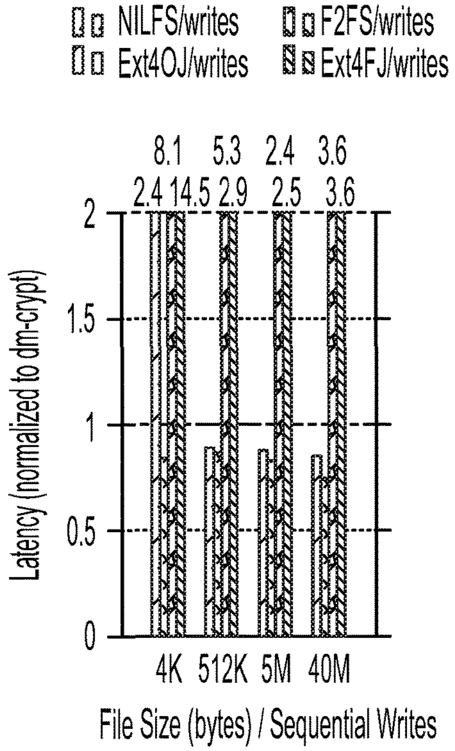


FIG. 8B

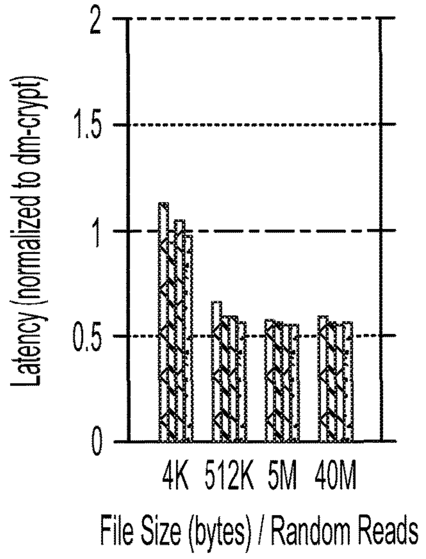


FIG. 8C

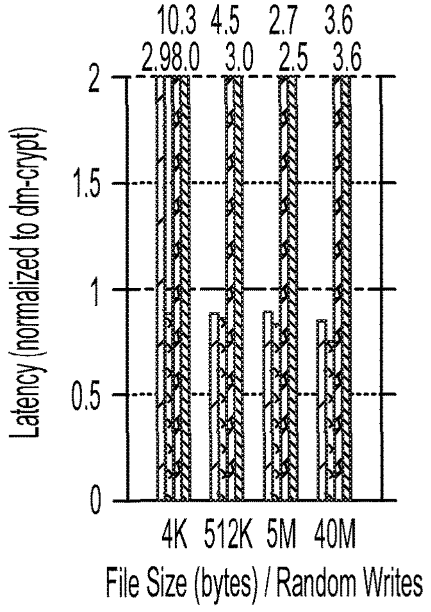


FIG. 8D

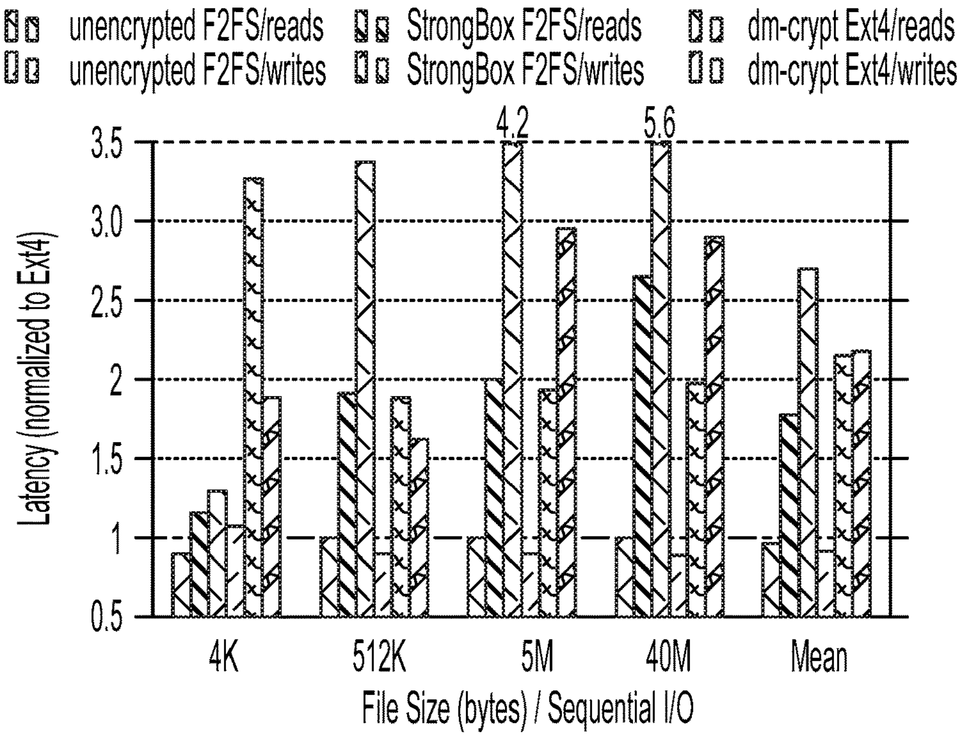

unencrypted F2FS/reads
StrongBox F2FS/reads
dm-crypt Ext4/reads
unencrypted F2FS/writes
StrongBox F2FS/writes
dm-crypt Ext4/writes
Latency (normalized to Ext4)
3.5
3.0
2.5
2
1.5
1
0.5
4.2
5.6
4K
512K
5M
40M
Mean
File Size (bytes) / Sequential I/O

FIG. 9A

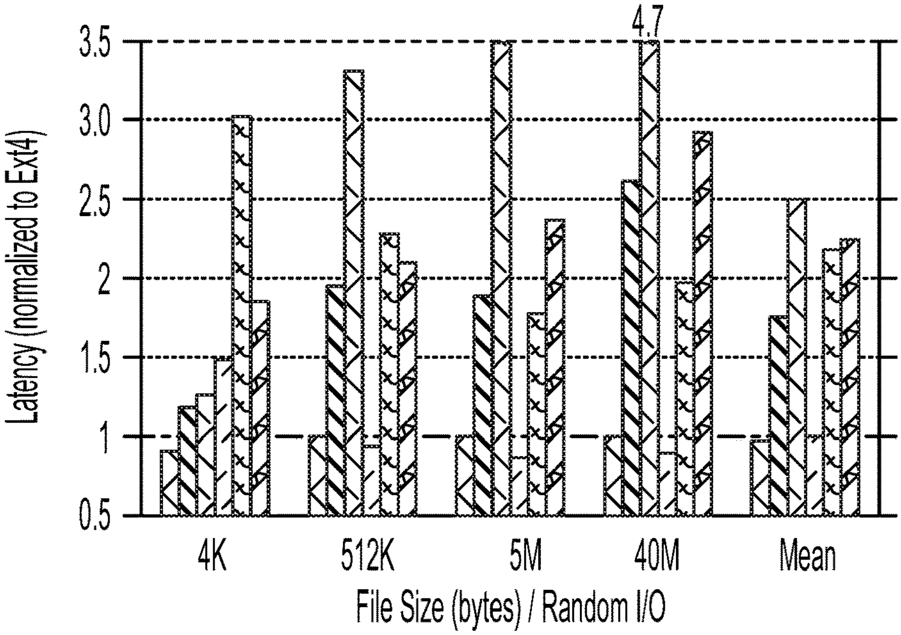

Latency (normalized to Ext4)
3.5
3.0
2.5
2
1.5
1
0.5
4.7
4K
512K
5M
40M
Mean
File Size (bytes) / Random I/O

FIG. 9B

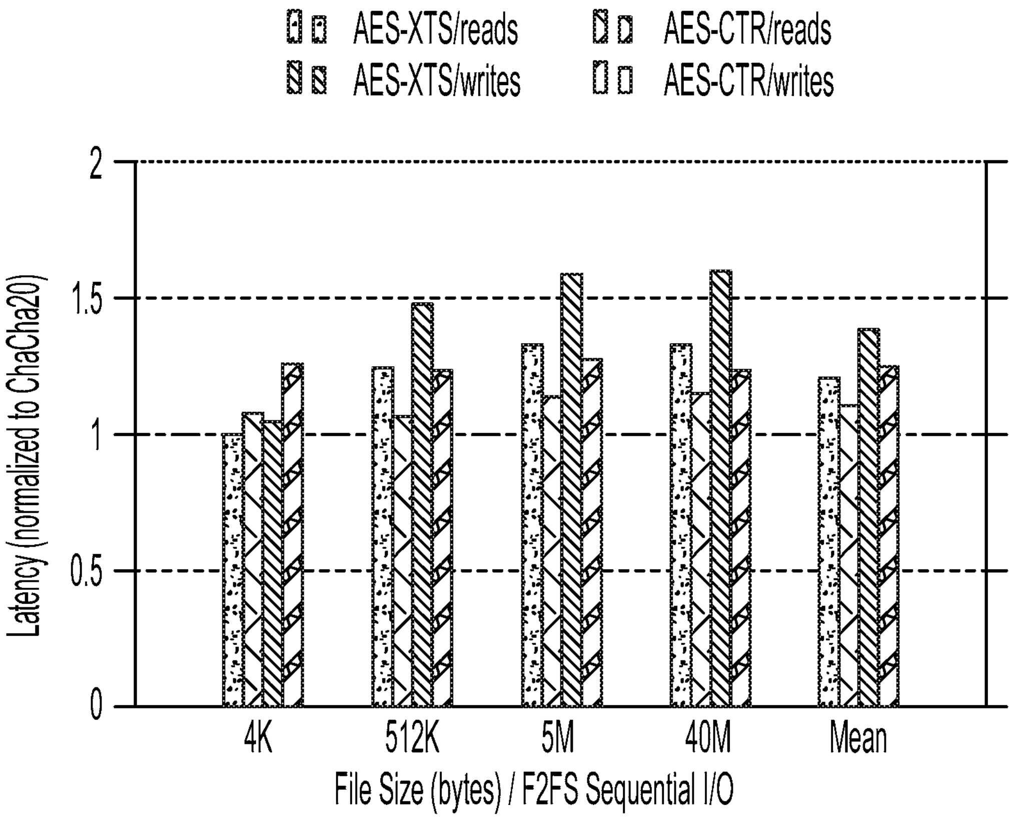

AES-XTS/reads
AES-CTR/reads
AES-XTS/writes
AES-CTR/writes
Latency (normalized to ChaCha20)
2
1.5
1
0.5
0
4K
512K
5M
40M
Mean
File Size (bytes) / F2FS Sequential I/O

FIG. 10

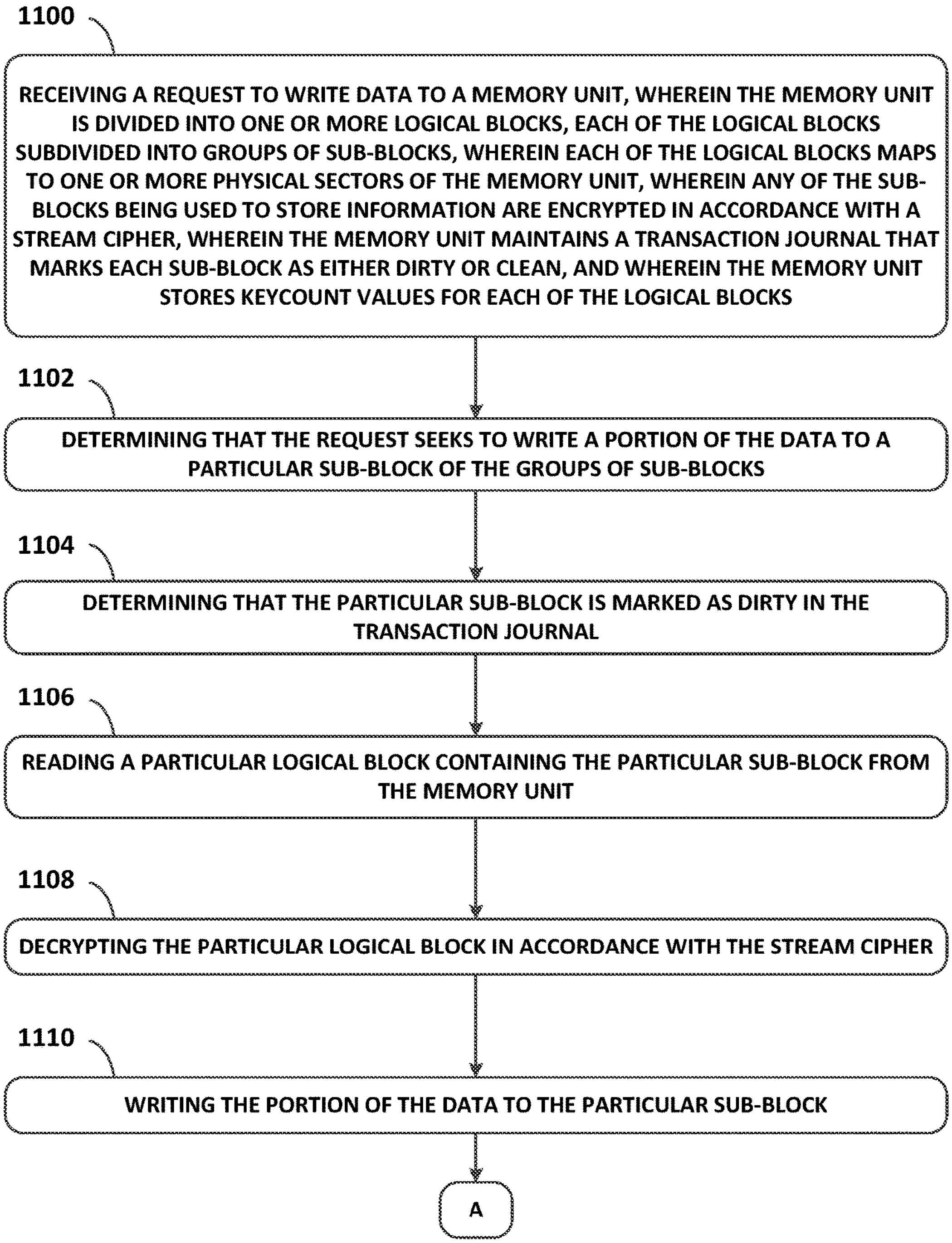

1100
RECEIVING A REQUEST TO WRITE DATA TO A MEMORY UNIT, WHEREIN THE MEMORY UNIT IS DIVIDED INTO ONE OR MORE LOGICAL BLOCKS, EACH OF THE LOGICAL BLOCKS SUBDIVIDED INTO GROUPS OF SUB-BLOCKS, WHEREIN EACH OF THE LOGICAL BLOCKS MAPS TO ONE OR MORE PHYSICAL SECTORS OF THE MEMORY UNIT, WHEREIN ANY OF THE SUB-BLOCKS BEING USED TO STORE INFORMATION ARE ENCRYPTED IN ACCORDANCE WITH A STREAM CIPHER, WHEREIN THE MEMORY UNIT MAINTAINS A TRANSACTION JOURNAL THAT MARKS EACH SUB-BLOCK AS EITHER DIRTY OR CLEAN, AND WHEREIN THE MEMORY UNIT STORES KEYCOUNT VALUES FOR EACH OF THE LOGICAL BLOCKS
1102
DETERMINING THAT THE REQUEST SEEKS TO WRITE A PORTION OF THE DATA TO A PARTICULAR SUB-BLOCK OF THE GROUPS OF SUB-BLOCKS
1104
DETERMINING THAT THE PARTICULAR SUB-BLOCK IS MARKED AS DIRTY IN THE TRANSACTION JOURNAL
1106
READING A PARTICULAR LOGICAL BLOCK CONTAINING THE PARTICULAR SUB-BLOCK FROM THE MEMORY UNIT
1108
DECRYPTING THE PARTICULAR LOGICAL BLOCK IN ACCORDANCE WITH THE STREAM CIPHER
1110
WRITING THE PORTION OF THE DATA TO THE PARTICULAR SUB-BLOCK
A

FIG. 11A

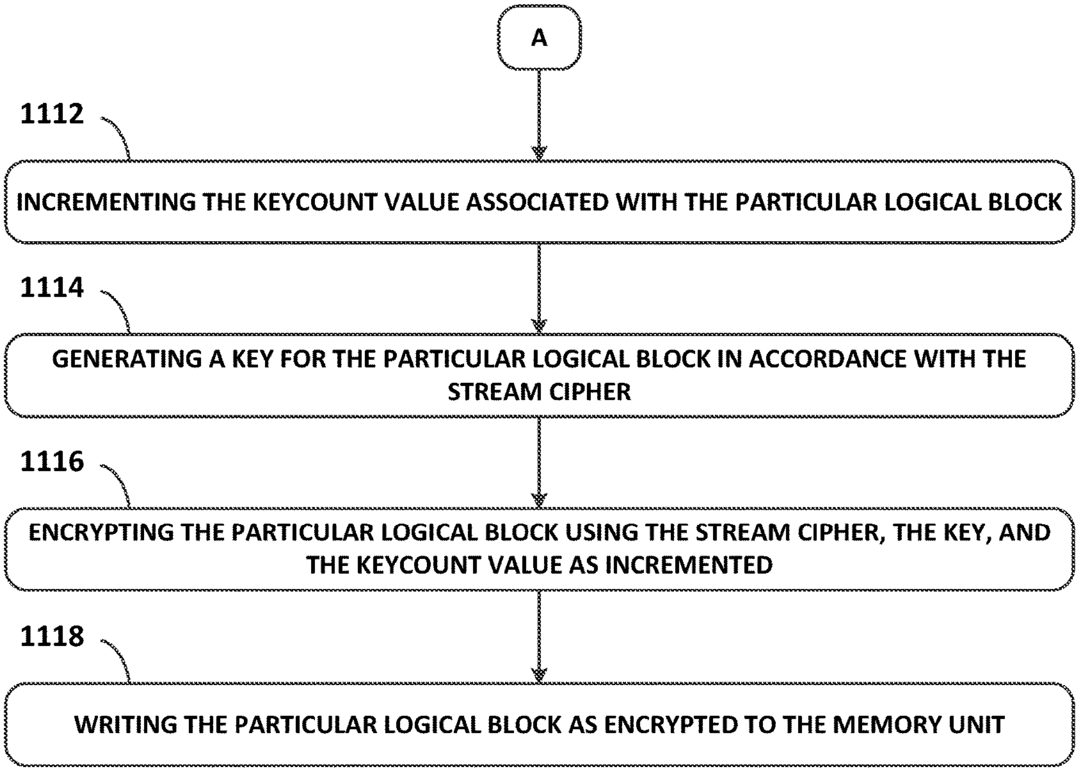

A
1112
INCREMENTING THE KEYCOUNT VALUE ASSOCIATED WITH THE PARTICULAR LOGICAL BLOCK
1114
GENERATING A KEY FOR THE PARTICULAR LOGICAL BLOCK IN ACCORDANCE WITH THE STREAM CIPHER
1116
ENCRYPTING THE PARTICULAR LOGICAL BLOCK USING THE STREAM CIPHER, THE KEY, AND THE KEYCOUNT VALUE AS INCREMENTED
1118
WRITING THE PARTICULAR LOGICAL BLOCK AS ENCRYPTED TO THE MEMORY UNIT

FIG. 11B

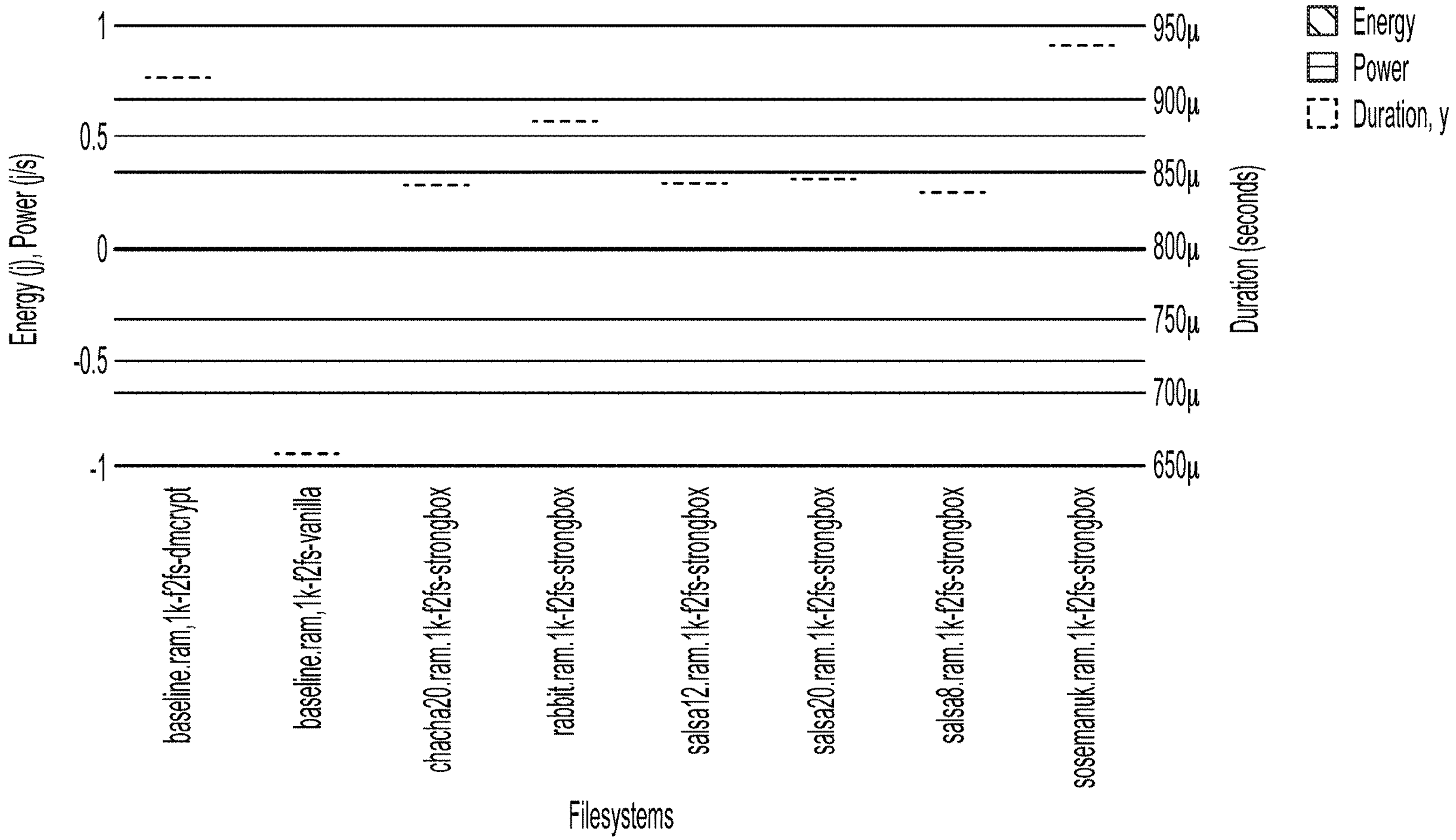

1
0.5
0
-0.5
-1
Energy (j), Power (j/s)
950μ
900μ
850μ
800μ
750μ
700μ
650μ
Duration (seconds)
Energy
Power
Duration, y
baseline.ram,1k-f2fs-dmcrypt
baseline.ram,1k-f2fs-vanilla
chacha20.ram.1k-f2fs-strongbox
rabbit.ram.1k-f2fs-strongbox
salsa12.ram.1k-f2fs-strongbox
salsa20.ram.1k-f2fs-strongbox
salsa8.ram.1k-f2fs-strongbox
sosemanuk.ram.1k-f2fs-strongbox
Filesystems

FIG. 12A

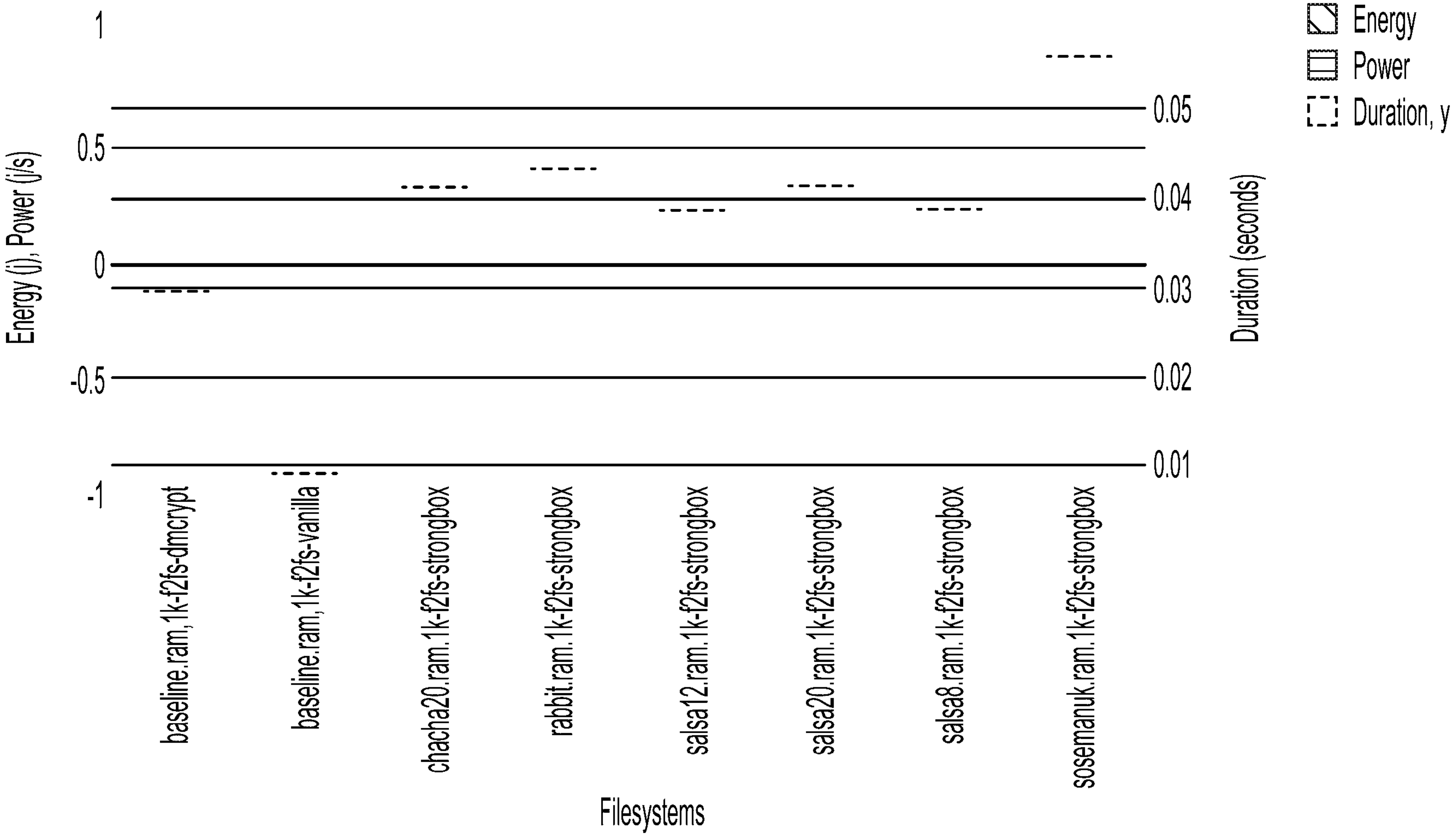

Energy (j), Power (j/s)
1
0.5
0
-0.5
-1
Duration (seconds)
0.05
0.04
0.03
0.02
0.01
Energy
Power
Duration, y
baseline.ram,1k-f2fs-dmcrypt
baseline.ram,1k-f2fs-vanilla
chacha20.ram.1k-f2fs-strongbox
rabbit.ram.1k-f2fs-strongbox
salsa12.ram.1k-f2fs-strongbox
salsa20.ram.1k-f2fs-strongbox
salsa8.ram.1k-f2fs-strongbox
sosemanuk.ram.1k-f2fs-strongbox
Filesystems

FIG. 12B

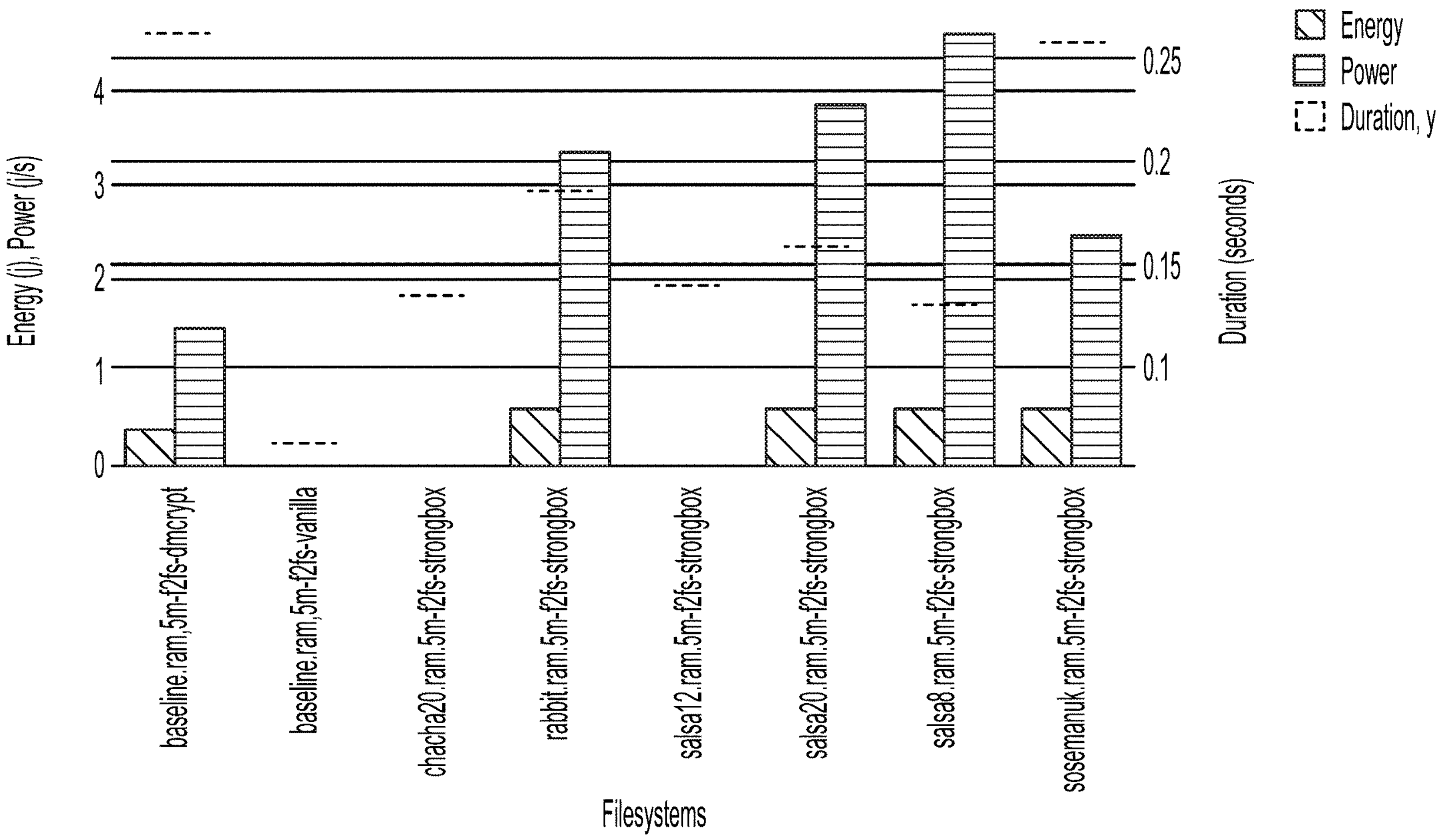

Energy (j), Power (j/s)
0
1
2
3
4
0.1
0.15
0.2
0.25
Duration (seconds)
baseline.ram,5m-f2fs-dmcrypt
baseline.ram,5m-f2fs-vanilla
chacha20.ram.5m-f2fs-strongbox
rabbit.ram.5m-f2fs-strongbox
salsa12.ram.5m-f2fs-strongbox
salsa20.ram.5m-f2fs-strongbox
salsa8.ram.5m-f2fs-strongbox
sosemanuk.ram.5m-f2fs-strongbox
Filesystems
Energy
Power
Duration, y

FIG. 12C

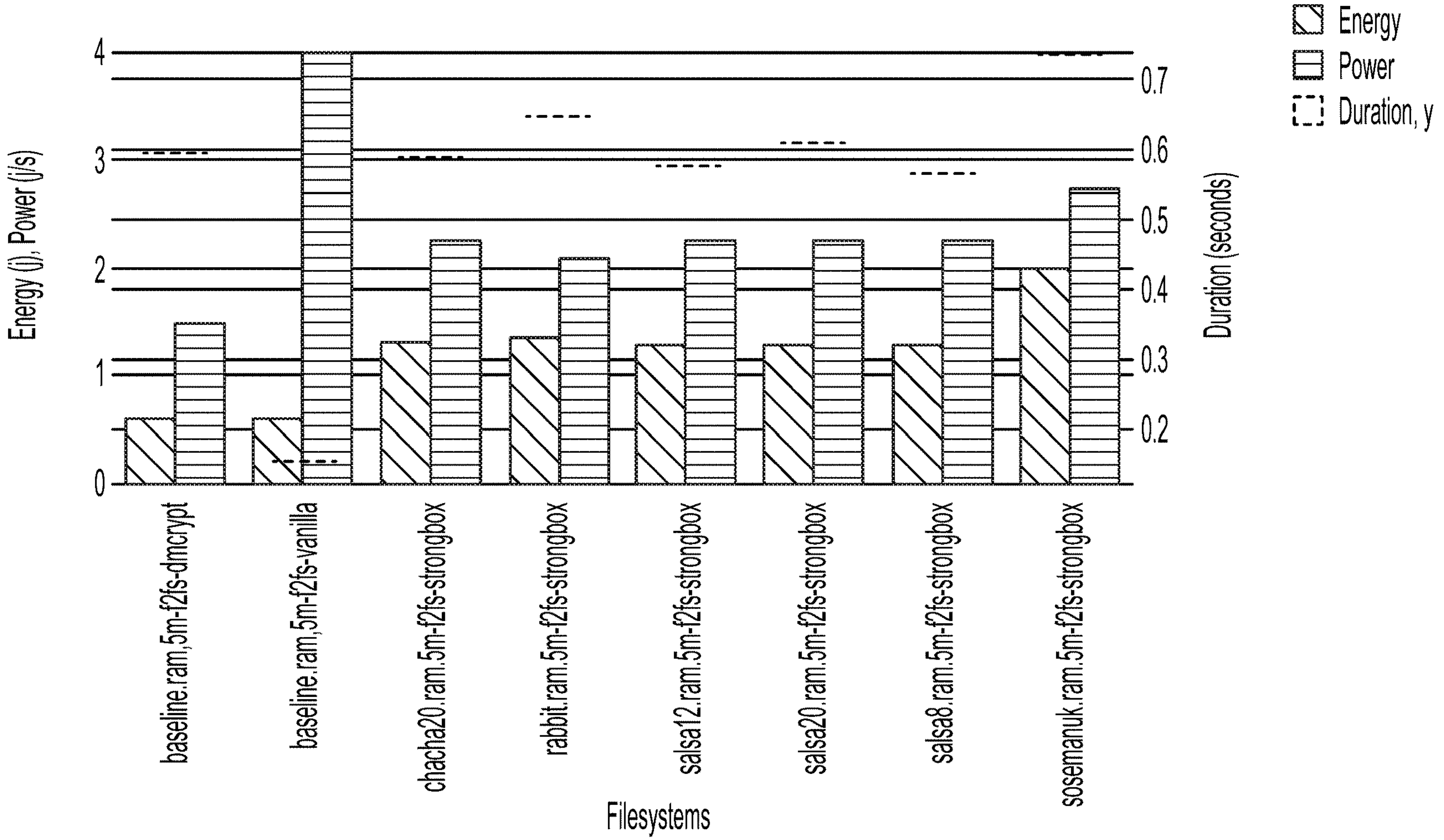

Energy (j), Power (j/s)
0
1
2
3
4
Duration (seconds)
0.2
0.3
0.4
0.5
0.6
0.7
Energy
Power
Duration, y
baseline.ram,5m-f2fs-dmcrypt
baseline.ram,5m-f2fs-vanilla
chacha20.ram.5m-f2fs-strongbox
rabbit.ram.5m-f2fs-strongbox
salsa12.ram.5m-f2fs-strongbox
salsa20.ram.5m-f2fs-strongbox
salsa8.ram.5m-f2fs-strongbox
sosemanuk.ram.5m-f2fs-strongbox
Filesystems

FIG. 12D

STREAM CIPHERS FOR DIGITAL STORAGE ENCRYPTION

CROSS-REFERENCE TO RELATED APPLICATION

[0001] This application claims priority to U.S. provisional patent application No. 62/625,148, filed Feb. 1, 2018, which is hereby incorporated by reference in its entirety.

BACKGROUND

[0002] Full drive encryption (FDE) protects the privacy of data at rest. For mobile devices, maintaining data privacy is especially important, as these devices contain sensitive personal and financial data yet are easily lost or stolen. The current standard for securing data at rest is to use the Advanced Encryption Standard (AES) cipher in XOR-Encrypt-XOR Tweaked CodeBook with Ciphertext Stealing (XTS) mode (referred to herein as AES-XTS). However, employing AES-XTS can increase read/write latency by up to 3-5 times compared to unencrypted storage.

[0003] Authenticated encryption using stream ciphers, such as ChaCha20, is faster than using AES. Indeed, some entities now use a stream cipher for Secure HyperText Transport Protocol (HTTPS) connections to obtain better performance. Stream ciphers are not used for FDE, however, for reasons of confidentiality and performance. Regarding confidentiality, when applied naively to stored data, stream ciphers are vulnerable to attacks, including many-time pad and rollback attacks, that reveal plaintext by overwriting a secure storage location using the same key. Further, it has been assumed that adding the metadata required to resist these attacks would ruin the stream cipher's performance advantage. Thus, the conventional wisdom is that FDE necessarily incurs the overhead of AES-XTS or a similar technique.

SUMMARY

[0004] Two technological shifts in mobile device and other hardware overturn this conventional wisdom, enabling confidential, high-performance storage with stream ciphers. First, these devices commonly use Log-structured File Systems (LFSs) or functionally equivalent constructions in hardware/firmware and/or software to increase the lifetime of their flash memory devices (e.g. solid state drives (SSDs)). Second, mobile devices now support trusted hardware, such as Trusted Execution Environments (TEEs) and secure storage areas. The use of LFSs limits overwrites to the same drive sectors; most writes are simply appended to a log, reducing the opportunity for attacks based on overwrites. The presence of secure hardware means that drive encryption modules have access to persistent, monotonically increasing counters that can be used to prevent rollback attacks when overwrites do occur.

[0005] Given these trends, the embodiments herein introduce a new method for securing data at rest. These embodiments may be implemented as a drop-in replacement for AES-XTS-backed FDE modules (i.e., no interface changes). The primary challenge is that even with an LFS running above an SSD, filesystem blocks will occasionally be overwritten; e.g., by segment cleaning or garbage collection. The embodiments overcome this challenge by using a fast stream cipher for confidentiality and performance with MAC tags and a secure, persistent hardware counter to ensure integrity and prevent attacks. The result is a system design enabling the first confidential, high-performance drive encryption based on a stream cipher. Nonetheless, certain types of block ciphers that mimic aspects of stream ciphers or exhibit behavior that is in some ways similar to that of stream ciphers (e.g., AES-CTR) could advantageously employ the embodiments herein.

[0006] Experimental results establish that the embodiments disclosed herein, when compared to AES-XTS implementations, reduce read latencies by as much as a factor of 2 (with a 1.6× mean improvement), and achieve near parity or provide an improvement in observed write latencies in the majority of benchmarks (a 1.2× mean improvement). This write performance is attained despite having to maintain more metadata.

[0007] Furthermore, these advances are accompanied by a stronger integrity guarantee than AES-XTS. Whereas XTS mode only randomizes plaintext when the ciphertext is altered the embodiments herein provide the security of standard authenticated encryption.

[0008] Accordingly, a first example embodiment may involve receiving a request to write data to a memory unit. The memory unit may be divided into one or more logical blocks, each of the logical blocks subdivided into groups of sub-blocks. Each of the logical blocks maps to one or more physical sectors of the memory unit. Any of the sub-blocks being used to store information are encrypted in accordance with a stream cipher. The memory unit maintains a transaction journal that marks each sub-block as either dirty or clean. The memory unit stores keycount values for each of the logical blocks. A cryptography software module may perform steps including: determining that the request seeks to write a portion of the data to a particular sub-block of the groups of sub-blocks, determining that the particular sub-block is marked as dirty in the transaction journal, reading a particular logical block containing the particular sub-block from the memory unit, decrypting the particular logical block in accordance with the stream cipher, writing the portion of the data to the particular sub-block, incrementing the keycount value associated with the particular logical block, generating a key for the particular logical block in accordance with the stream cipher, encrypting the particular logical block using the stream cipher, the key, and the keycount value as incremented, and writing the particular logical block as encrypted to the memory unit.

[0009] In a second example embodiment, a method may be used to perform operations in accordance with the first example embodiment.

[0010] In a third example embodiment, an article of manufacture may include a non-transitory computer-readable medium, having stored thereon program instructions that, upon execution by a computing system, cause the computing system to perform operations in accordance with the first example embodiment.

[0011] In a fourth example embodiment, a system may include various means for carrying out each of the operations of the first example embodiment.

[0012] These as well as other embodiments, aspects, advantages, and alternatives will become apparent to those of ordinary skill in the art by reading the following detailed description, with reference where appropriate to the accompanying drawings. Further, this summary and other descriptions and figures provided herein are intended to illustrate embodiments by way of example only and, as such, that numerous variations are possible. For instance, structural elements and process steps can be rearranged, combined, distributed, eliminated, or otherwise changed, while remaining within the scope of the embodiments as claimed.

BRIEF DESCRIPTION OF THE DRAWINGS

[0013] FIG. **1** illustrates a schematic drawing of a computing device, in accordance with example embodiments.

[0014] FIG. **2** illustrates a schematic drawing of a server device cluster, in accordance with example embodiments.

[0015] FIG. **3** depicts performance improvements of the embodiments herein over traditional file system structures.

[0016] FIG. **4** is a block diagram representing an FDE mechanism, in accordance with example embodiments.

[0017] FIG. **5** is a block diagram representing a backing store of the FDE mechanism, in accordance with example embodiments.

[0018] FIG. **6**A depicts an algorithm for handling a request to read from encrypted storage, in accordance with example embodiments.

[0019] FIG. **6**B depicts an algorithm for handling a request to write to encrypted storage, in accordance with example embodiments.

[0020] FIG. **6**C depicts an algorithm for rekeying, in accordance with example embodiments.

[0021] FIG. **7**A illustrates performance results, in accordance with example embodiments.

[0022] FIG. **7**B illustrates performance results, in accordance with example embodiments.

[0023] FIG. **8**A illustrates performance results, in accordance with example embodiments.

[0024] FIG. **8**B illustrates performance results, in accordance with example embodiments.

[0025] FIG. **8**C illustrates performance results, in accordance with example embodiments.

[0026] FIG. **8**D illustrates performance results, in accordance with example embodiments.

[0027] FIG. **9**A illustrates performance results, in accordance with example embodiments.

[0028] FIG. **9**B illustrates performance results, in accordance with example embodiments.

[0029] FIG. **10** illustrates performance results, in accordance with example embodiments.

[0030] FIGS. **11**A and **11**B depict a flow chart, in accordance with example embodiments.

[0031] FIG. **12**A illustrates performance results, in accordance with example embodiments.

[0032] FIG. **12**B illustrates performance results, in accordance with example embodiments.

[0033] FIG. **12**C illustrates performance results, in accordance with example embodiments.

[0034] FIG. **12**D illustrates performance results, in accordance with example embodiments.

DETAILED DESCRIPTION

[0035] Example methods, devices, and systems are described herein. It should be understood that the words "example" and "exemplary" are used herein to mean "serving as an example, instance, or illustration." Any embodiment or feature described herein as being an "example" or "exemplary" is not necessarily to be construed as preferred or advantageous over other embodiments or features unless stated as such. Thus, other embodiments can be utilized and other changes can be made without departing from the scope of the subject matter presented herein.

[0036] Accordingly, the example embodiments described herein are not meant to be limiting. It will be readily understood that the aspects of the present disclosure, as generally described herein, and illustrated in the figures, can be arranged, substituted, combined, separated, and designed in a wide variety of different configurations. For example, the separation of features into "client" and "server" components may occur in a number of ways.

[0037] Further, unless context suggests otherwise, the features illustrated in each of the figures may be used in combination with one another. Thus, the figures should be generally viewed as component aspects of one or more overall embodiments, with the understanding that not all illustrated features are necessary for each embodiment.

[0038] Additionally, any enumeration of elements, blocks, or steps in this specification or the claims is for purposes of clarity. Thus, such enumeration should not be interpreted to require or imply that these elements, blocks, or steps adhere to a particular arrangement or are carried out in a particular order.

I. Example Computing Devices and Cloud-Based Computing Environments

[0039] The following embodiments describe architectural and operational aspects of example computing devices and systems that may employ the disclosed FDE implementations, as well as the features and advantages thereof.

[0040] FIG. **1** is a simplified block diagram exemplifying a computing device **100**, illustrating some of the components that could be included in a computing device arranged to operate in accordance with the embodiments herein. Computing device **100** could be a client device (e.g., a device actively operated by a user), a server device (e.g., a device that provides computational services to client devices), or some other type of computational platform. Some server devices may operate as client devices from time to time in order to perform particular operations, and some client devices may incorporate server features.

[0041] In this example, computing device **100** includes processor **102**, memory **104**, network interface **106**, and an input/output unit **108**, all of which may be coupled by a system bus **110** or a similar mechanism. In some embodiments, computing device **100** may include other components and/or peripheral devices (e.g., detachable storage, printers, and so on).

[0042] Processor **102** may be one or more of any type of computer processing element, such as a central processing unit (CPU), a co-processor (e.g., a mathematics, graphics, or encryption co-processor), a digital signal processor (DSP), a network processor, and/or a form of integrated circuit or controller that performs processor operations. In some cases, processor **102** may be one or more single-core processors. In other cases, processor **102** may be one or more multi-core processors with multiple independent processing units. Processor **102** may also include register memory for temporarily storing instructions being executed and related data, as well as cache memory for temporarily storing recently-used instructions and data.

[0043] Memory **104** may be any form of computer-usable memory, including but not limited to random access memory (RAM), read-only memory (ROM), and non-volatile memory. This may include flash memory, hard disk drives, solid state drives, re-writable compact discs (CDs), re-writable digital video discs (DVDs), and/or tape storage, as just a few examples. Computing device **100** may include fixed memory as well as one or more removable memory units, the latter including but not limited to various types of secure digital (SD) cards. Thus, memory **104** represents both main memory units, as well as long-term storage. Other types of memory may include biological memory.

[0044] Memory **104** may store program instructions and/or data on which program instructions may operate. By way of example, memory **104** may store these program instructions on a non-transitory, computer-readable medium, such that the instructions are executable by processor **102** to carry out any of the methods, processes, or operations disclosed in this specification or the accompanying drawings.

[0045] As shown in FIG. **1**, memory **104** may include firmware **104**A, kernel **104**B, and/or applications **104**C. Firmware **104**A may be program code used to boot or otherwise initiate some or all of computing device **100**. Kernel **104**B may be an operating system, including modules for memory management, scheduling and management of processes, input/output, and communication. Kernel **104**B may also include device drivers that allow the operating system to communicate with the hardware modules (e.g., memory units, networking interfaces, ports, and busses), of computing device **100**. Applications **104**C may be one or more user-space software programs, such as web browsers or email clients, as well as any software libraries used by these programs. Memory **104** may also store data used by these and other programs and applications.

[0046] Network interface **106** may take the form of one or more wireline interfaces, such as Ethernet (e.g., Fast Ethernet, Gigabit Ethernet, and so on). Network interface **106** may also support communication over one or more non-Ethernet media, such as coaxial cables or power lines, or over wide-area media, such as Synchronous Optical Networking (SONET) or digital subscriber line (DSL) technologies. Network interface **106** may additionally take the form of one or more wireless interfaces, such as IEEE 802.11 (Wifi), BLUETOOTH®, global positioning system (GPS), or a wide-area wireless interface. However, other forms of physical layer interfaces and other types of standard or proprietary communication protocols may be used over network interface **106**. Furthermore, network interface **106** may comprise multiple physical interfaces. For instance, some embodiments of computing device **100** may include Ethernet, BLUETOOTH®, and Wifi interfaces.

[0047] Input/output unit **108** may facilitate user and peripheral device interaction with example computing device **100**. Input/output unit **108** may include one or more types of input devices, such as a keyboard, a mouse, a touch screen, and so on. Similarly, input/output unit **108** may include one or more types of output devices, such as a screen, monitor, printer, and/or one or more light emitting diodes (LEDs). Additionally or alternatively, computing device **100** may communicate with other devices using a universal serial bus (USB) or high-definition multimedia interface (HDMI) port interface, for example.

[0048] In some embodiments, one or more instances of computing device **100** may be deployed to support a clustered architecture. The exact physical location, connectivity, and configuration of these computing devices may be unknown and/or unimportant to client devices. Accordingly, the computing devices may be referred to as "cloud-based" devices that may be housed at various remote data center locations.

[0049] FIG. **2** depicts a cloud-based server cluster **200** in accordance with example embodiments. In FIG. **2**, operations of a computing device (e.g., computing device **100**) may be distributed between server devices **202**, data storage **204**, and routers **206**, all of which may be connected by local cluster network **208**. The number of server devices **202**, data storages **204**, and routers **206** in server cluster **200** may depend on the computing task(s) and/or applications assigned to server cluster **200**.

[0050] For example, server devices **202** can be configured to perform various computing tasks of computing device **100**. Thus, computing tasks can be distributed among one or more of server devices **202**. To the extent that these computing tasks can be performed in parallel, such a distribution of tasks may reduce the total time to complete these tasks and return a result. For purpose of simplicity, both server cluster **200** and individual server devices **202** may be referred to as a "server device." This nomenclature should be understood to imply that one or more distinct server devices, data storage devices, and cluster routers may be involved in server device operations.

[0051] Data storage **204** may be data storage arrays that include drive array controllers configured to manage read and write access to groups of hard disk drives and/or solid state drives. The drive array controllers, alone or in conjunction with server devices **202**, may also be configured to manage backup or redundant copies of the data stored in data storage **204** to protect against drive failures or other types of failures that prevent one or more of server devices **202** from accessing units of cluster data storage **204**. Other types of memory aside from drives may be used.

[0052] Routers **206** may include networking equipment configured to provide internal and external communications for server cluster **200**. For example, routers **206** may include one or more packet-switching and/or routing devices (including switches and/or gateways) configured to provide (i) network communications between server devices **202** and data storage **204** via cluster network **208**, and/or (ii) network communications between the server cluster **200** and other devices via communication link **210** to network **212**.

[0053] Additionally, the configuration of cluster routers **206** can be based at least in part on the data communication requirements of server devices **202** and data storage **204**, the latency and throughput of the local cluster network **208**, the latency, throughput, and cost of communication link **210**, and/or other factors that may contribute to the cost, speed, fault-tolerance, resiliency, efficiency and/or other design goals of the system architecture.

[0054] As a possible example, data storage **204** may include any form of database, such as a structured query language (SQL) database. Various types of data structures may store the information in such a database, including but not limited to tables, arrays, lists, trees, and tuples. Furthermore, any databases in data storage **204** may be monolithic or distributed across multiple physical devices.

[0055] Server devices **202** may be configured to transmit data to and receive data from cluster data storage **204**. This transmission and retrieval may take the form of SQL queries or other types of database queries, and the output of such queries, respectively. Additional text, images, video, and/or audio may be included as well. Furthermore, server devices

202 may organize the received data into web page representations. Such a representation may take the form of a markup language, such as the hypertext markup language (HTML), the extensible markup language (XML), or some other standardized or proprietary format. Moreover, server devices **202** may have the capability of executing various types of computerized scripting languages, such as but not limited to Perl, Python, PHP Hypertext Preprocessor (PHP), Active Server Pages (ASP), JavaScript, and so on. Computer program code written in these languages may facilitate the providing of web pages to client devices, as well as client device interaction with the web pages.

II. Using Stream Ciphers with FDE

[0056] One of the motivations for the embodiments herein is the speed of stream ciphers compared to other types of ciphers, such as block ciphers. In general, block ciphers operate on fixed-length blocks of bits, each encoded using the same transformation and symmetric key. AES is an example of a block cipher. For FDE applications, various modifications to block ciphers has been proposed and put into use. As an example, XTS mode uses cipher block chaining (CBC) to combine (typically by way of an XOR operation) the previous block's ciphertext with the current block's plaintext. Thus, the decryption of one block depends on the decryption of all previous blocks. XTS mode also used different keys for encryption of the initial block and later blocks of the plaintext. In practice, however, XTS mode does not support detection of tampering in and of itself, and can be susceptible to traffic analysis, replay and randomization attacks.

[0057] In contrast, a stream cipher involves plaintext digits being combined with a pseudorandom cipher digit stream (referred to as a keystream). Each plaintext digit is encrypted with the corresponding digit of the keystream, to give a digit of the resulting ciphertext stream. Thus, encryption of each digit is dependent on the current state of the cipher (in some variations, the encryption may also depend upon previously-processed ciphertext digits). In practice, a digit is typically a bit and the combining operation is typically an XOR. The keystream can be generated serially from a random seed value using digital shift registers. The seed value serves as the cryptographic key for decrypting the ciphertext stream. To be secure, the keystream should be virtually indistinguishable from random noise.

[0058] One of the main advantages of stream ciphers over other types of ciphers is speed. An Exynos Octa processor with an ARM big.LITTLE architecture was used to compare AES-XTS to the stream cipher ChaCha20+Polyl305. Particularly, 250 megabytes of randomly-generated bits were encrypted and decrypted three times, and the median time for each of encryption and decryption was recorded. Use of the stream cipher resulted in a 2.7× reduction of run time, as illustrated in FIG. **3**.

[0059] Still, stream ciphers are not designed to encrypt data at rest. In a naive implementation of FDE with a stream cipher, overwriting the same memory location with the same key would trivially allow an attacker to recover the secret key. Thus, stream ciphers may be better suited for encrypting block devices using Log-structured File Systems (LFSs).

[0060] A traditional file system writes files to a storage medium in order to leverage spatial and temporal locality-of-reference, as well as to be able to make in-place changes to data structures. On the other hand, an LFS divides the storage medium into segments and writes files to each segment in the form of logs. Rather than overwrite an existing location, the LFS places new writes at the end of the log, and reclaims storage space through garbage collection on older log entries. As a result, multiple versions of a file can be supported, and storage integrity activities after a crash are simpler.

[0061] Since LFSs are designed to append data to the end of a log rather than overwrite data, they are a good fit for stream ciphers, as it is highly unlikely that the same memory location would be overwritten using the same key. In practice, some overwrites occur; e.g., in metadata, but they are small in number during normal execution. Notably, although some of the embodiments herein and the associated experimental results assume an LFS, the embodiments can be used with and are likely to produce improvements when employed on other types of file systems as well.

[0062] As an example, 800 megabytes of random data was written directly to a memory module using four different file systems: Ext4, LogFS, NILFS, and F2FS. Ext4 is a journaling file system, whereas LogFS, NILFS, and F2FS are LFSs. A journaling file system provides a separate log for tracking changes in files, but overwrites files in place.

TABLE 1

File System	Total Write Operations	Overwrites
Ext4	16,756	10,787
LogFS	4,244	32
NILFS	4,199	24
F2FS	2,107	2

[0063] The number of total writes to the underlying block device and the number of times data was overwritten for each file system was counted and is shown in Table 1. In the results, Ext4 exhibits the highest number of writes, but many of those are small writes for book-keeping purposes. Ext4 also has the largest number of overwrites, as almost 65% of the writes are to a previously written location in the backing store (here, the backing store is the underlying memory device). In contrast, all three log-structured file systems have very few overwrites.

[0064] Use of a stream cipher has the advantage of being more than twice as fast as AES-XTS, while providing the same confidentiality guarantee. The problem is that the stream cipher is not secure if the same key is used to overwrite the same storage location. Fortunately, the LFSs rarely overwrite the same location. This makes stream ciphers a good candidate for securing data stored in an LFS. Nonetheless, overwrites to an LFS do occur. While Table 1 shows overwrites are rare during normal operation, they will occur when garbage collecting the LFS. Thus, the embodiments here may use metadata to track writes and ensure that data is re-keyed if overwrites occur.

[0065] Overall, there are three main challenges to replacing AES with a stream cipher for FDE: (i) tracking writes to the memory to ensure that the same location is not overwritten with the same key, (ii) ensuring that the metadata that tracks these writes is secure and not subject to leaks or rollback attacks, (iii) accomplishing these tasks efficiently so that the performance advantage of the stream cipher is maintained.

[0066] These challenges can be met by using a secure, persistent counter supported in modern mobile hardware; e.g., for limiting password attempts. This counter can track writes, and thus versions of the encrypted data. If an attacker tried to roll back the file system to overwrite the same location with the same key, the implementation detects that the local version number is out of sync with the global version number stored in the secure counter. In that case, the system refuses to initialize, and the attack fails. The use of the hardware-supported secure counter significantly raises the bar when it comes to rollback attacks, requiring a costly and non-discrete physical attack on the hardware itself to be effective. Nonetheless, the actual structure of the metadata required to track writes and maintain integrity is more complicated than simply implementing a counter and is described in detail below.

[0067] An additional challenge is that of crash recovery. The embodiments herein rely on the overlying filesystem to manage data recovery in the event of a crash that leaves user data in an inconsistent state. Metadata recovery is addressed after a crash by giving the root user the option to accept the current metadata state as the new consistent state, i.e., "force mounting" the filesystem. An attacker might try to take advantage of this feature by modifying the memory, forcing an inconsistent state, and hoping the root user will ignore it and force mount the system anyway. The embodiments defend against this attack by preventing force mounts when the metadata state is wildly inconsistent with the global version counter. Otherwise, the root user is warned if they attempt a force mount. Thus, attacking by forcing a crash can only be successful if the attacker also has root permission, in which case security is already compromised.

III. Example Architecture

[0068] The embodiments herein act as a translation layer placed between the disk and the operating system. They provide confidentiality and integrity guarantees while mitigating performance loss due to metadata management overhead. This is accomplished by leveraging the speed of stream ciphers over the AES block cipher and taking advantage of the append-mostly nature of Log-Structured Filesystems (LFS) and modern Flash Translation Layers (FTL).

[0069] Hence, there are several locations where the implementation could be positioned in the system stack. It could be integrated into an LFS filesystem module itself, e.g., F2FS, specifically leveraging the flexibility of the Virtual Filesystem Switch (VFS). Alternatively or additionally, it could be implemented as an actual block device or virtual block device layered atop a physical block device (the latter is where the implementation described herein operates). Alternatively or additionally, it could even be implemented within the on-disk SSD controller managing the flash translation layer (scatter gather, garbage collection, wear-leveling, etc.).

[0070] FIG. **4** depicts an example design. The metadata is encapsulated in four components: an in-memory Merkle Tree and two disk-backed byte arrays, the keycount store and the transaction journal, and a persistent monotonic counter (implemented with the replay protected memory block, or RPMB). All four are integrated with the Cryptographic Driver, which handles data encryption, verification, and decryption during interactions with the underlying backing store. These interactions take place while fulfilling high-level I/O requests received from the overlying LFS. The Device Controller handles low-level I/O between the Cryptographic Driver and the backing store.

[0071] A. Backing Store Function and Layout

[0072] FIG. **5** depicts a possible backing store layout. In the body section of the backing store, application data is partitioned into a series of same-size logical blocks. These logical blocks are distinct from the concept of physical disk blocks, which are collections of one or more disk sectors. To make this distinction clear, the logical blocks are referred to as nuggets, marked NUG in FIG. **5**. Hence, a nugget consists of one or more physical disk blocks, depending on its configured size. Each nugget is subdivided into a constant number of sub-blocks referred to as flakes. The reason for these nugget/flake divisions are two-fold: (i) to limit the maximum length of any plaintext operated on by the cryptographic driver, decreasing the overhead incurred per I/O operation, and (ii) to track, detect, and handle overwrites.

[0073] When a request comes in to write to one or more flakes in a nugget, the affected flakes are marked "dirty." Here, the marking of dirty implies that another write to some portion of that flake would constitute an overwrite. If a new request comes in to write to one or more of those same flakes another time, a rekeying procedure over the entire nugget is triggered to safely overwrite the old data in those flakes. This rekeying procedure may be time consuming, adding to the overhead of overwrites.

TABLE 2

Header	Length	Description
VERSION	4 bytes	Specifies the version of the encryption software used to initialize the backing store.
SALT	16 bytes	The salt used in part to derive the global master secret.
MTRH	32 bytes	Hash of the Merkle Tree root.
TPMGLOBALVER	8 bytes	The monotonic global version count, in hardware-supported secure storage.
VERIFICATION	32 bytes	Used to determine if the key derived from a password is correct.
NUMNUGGETS	4 bytes	The number of nuggets contained by the backing store.
FLAKESPERNUGGET	4 bytes	The number of flakes per nugget.
FLAKESIZE	4 bytes	The size of each flake, in bytes.
INITIALIZED	1 byte	Used to determine whether the backing store has been properly initialized.
REKEYING	4 bytes	The index of the nugget in need of rekeying if there is a pending rekeying procedure.

[0074] The head of the backing store contains the metadata written to disk during initialization. These headers govern operation and are described in more detail in Table 2. After the headers, two byte arrays are stored in the Head section. One is an array of N 8-byte integer keycounts and one of N [P/8] byte transaction journal entries, where N is the number of nuggets and P is the number of flakes per nugget. The Rekeying Journal is stored at the end of the Head section. The rekeying journal is where nuggets and their associated metadata are transiently written, so that rekeying can be resumed in the event that it is interrupted.

[0075] B. Metadata-Aware Cryptographic Driver

[0076] The cryptographic driver coordinates the system's disparate components. Its primary function is to map incoming reads and writes to their proper destinations in the backing store, applying the chosen stream cipher and message authentication code to encrypt, verify, and decrypt data on the fly with consideration for metadata management.

[0077] When a read request is received, it is first partitioned into affected nuggets; i.e., a read that spans two nuggets is partitioned in half. For each nugget affected, the flakes touched by the request are determined. Then, the contents of those flakes are verified. If all the flakes are valid, whatever subset of data that was requested by the user is decrypted and returned. Algorithm 1 as shown in FIG. **6**A details the read operation.

[0078] Like reads, when a write request is received, the request is first partitioned with respect to affected nuggets. For each affected nugget, which flakes are touched by the request are determined. These flakes are checked if any are marked as dirty in the transaction journal. If one or more of them have been marked dirty, rekeying for these specific nuggets is triggered. Rekeying is detailed in Algorithm 3 in FIG. **6**C. Otherwise, the touched flakes are marked as dirty in the transaction journal. Then, the touched flakes are iterated over. For the first and last flakes touched by the write request, an internal read request is executed (Algorithm 1 in FIG. **6**A) to both obtain the flake data and verify that data with the Merkle Tree. Then, every touched flake is overwritten with the data from the requested operation, the Merkle Tree is updated to reflect this change, and the new flake data is written and encrypted. Then, all corresponding metadata is committed. Algorithm 2 in FIG. **6**B details the write operation.

[0079] Herein, a Merkle Tree may be referred to as a hash tree. In such a tree, each leaf node contains a cryptographic hash of a flake, and every non-leaf node contains a cryptographic hash of its child nodes. This allows efficient verification of large amounts of data. The cryptographic hash may be any one-way function that maps an input bit string (potentially of arbitrary size) to an output bit string (potentially of fixed size). Regardless, the embodiments herein are not limited to using Merkle Trees or hash trees, and other types of data verification mechanisms may be used. For instance, an SHA-based c-struct implementation, a Tiger tree, a Fletcher-based tree of pointers, other Merkle Tree variations, or any other algorithm that can unify the state of all tags such that any change is immediately evident can be used.

[0080] 1. Transaction Journal

[0081] An overwrite breaks the security guarantee offered by any stream cipher. To prevent this failure, the embodiments herein track incoming write requests to prevent overwrites. This tracking is done with the transaction journal of FIG. **4**.

[0082] The transaction journal consists of N [P/8]-byte bit vectors, where N is the number of nuggets and P is the number of flakes per nugget. A bit vector v contains at least P bits=$b_0b_1b_2$, . . . , b_{p-1}, . . . , with extra bits ignored. Each vector is associated with a nugget and each bit with a flake belonging to that nugget. When an incoming write request occurs, the corresponding bit vector is updated (set to 1) to reflect the new dirty state of those flakes.

[0083] The transaction journal is referenced during each write request, where it is updated to reflect the state of the nugget and checked to ensure the operation does not constitute an overwrite. If the operation does constitute an overwrite, a rekeying procedure is triggered for the entire nugget before safely completing the request.

[0084] 2. Merkle Tree

[0085] Tracking writes with the transaction journal may stymie a passive attacker by preventing explicit overwrites, but a sufficiently motivated active attacker could resort to all manner of cut-and-paste tactics with nuggets, flakes, and even blocks and sectors. If, for example, an attacker purposefully zeroed-out the transaction journal entry pertaining to a specific nugget in some out-of-band manner, such as when the system is shut down and then later re-initialized with the same backing store, the system would consider any successive incoming writes as if the nugget were in a completely clean state, even though it actually is not. This attack would force compromising overwrites. To prevent such attacks, it can be ensured that the backing store is always in a valid state. More concretely, there should be an integrity guarantee on top of a confidentiality guarantee.

[0086] The system uses a Message Authentication Code (MAC) algorithm and each flake's unique key to generate a per-flake MAC tag. Each tag is then appended to the Merkle Tree along with metadata. The transaction journal entries are handled specially in that the bit vectors are MACed and the result is appended to the Merkle Tree. This is done to save space.

[0087] 3. Keycount Store

[0088] To prevent a many-time pad attack, each nugget is assigned its own form of nonce referred to as a keycount. The keycount store in FIG. **4** represents a byte-array containing N 8-byte integer keycounts indexed to each nugget. Along with acting as the per-nugget nonce consumed by the stream cipher, the keycount is used to derive the per-flake unique subkeys used in MAC tag generation.

[0089] 4. Rekeying Procedure

[0090] When a write request would constitute an overwrite, the system triggers a rekeying process instead of executing the write normally. This rekeying process allows the write to proceed without causing a catastrophic confidentiality violation.

[0091] When rekeying begins, the nugget in question is loaded into memory and decrypted. The target data is written into its proper offset in this decrypted nugget. The nugget is then encrypted, this time with a different nonce (keycount+1), and written to the backing store, replacing the outdated nugget data. Algorithm 3 in FIG. **6**C details this procedure.

[0092] C. Defending Against Rollback Attacks

[0093] To prevent making overwrites, the status of each flake is tracked and overwrites trigger a rekeying procedure. Tracking flake status alone is not enough, however. An attacker could take a snapshot of the backing store in its current state and then easily rollback to a previously valid state. At this point, the attacker could have the system make writes that it does not recognize as overwrites.

[0094] With AES-XTS, the threat posed by rolling the backing store to a previously valid state is outside of its threat model. Despite this, data confidentiality guaranteed by AES-XTS holds in the event of a rollback, even if integrity is violated. The embodiments herein use a monotonic global version counter to detect rollbacks. When a rollback is detected, the system refuses to initialize unless forced, using root permission. Whenever a write request is completed, this global version counter is committed to the backing store, committed to secure hardware, and updated in the in-memory Merkle Tree.

[0095] D. Recovering From Inconsistent State

[0096] If the system is interrupted during operation, the backing store—consisting of user data and metadata—can be left in an inconsistent state. The system relies on the overlying filesystem (e.g., F2FS) to manage user-data recovery, which is what these filesystems are designed to do and do well. The system handles its own inconsistent metadata.

[0097] Let c be the value of the on-chip monotonic global version counter and d be the value of the on-disk global version counter header (TPMGLOBALVER). Consider the following cases.

[0098] Case 1:c==d and MTRH is consistent: The system is operating normally and will mount without issue.

[0099] Case 2:c<d or c==d but MTRH is inconsistent: Since the global version counter is updated before any write, this case cannot be reached unless the backing store was manipulated by an attacker. So, the system refuses to initialize and cannot be force mounted.

[0100] Case 3:c>d+1: Since the global version counter is updated once per write, this case cannot be reached unless the backing store was rolled back or otherwise manipulated by an attacker. In this case, the root user is warned and the system refuses to initialize and cannot be force mounted unless the MTRH is consistent. The root user can force mount if the root user initiated the rollback themselves, such as when recovering from a disk backup.

[0101] Case 4:c==d+1: In this case, the system likely crashed during a write, perhaps during an attempted rekeying. If the rekeying journal is empty or the system cannot complete the rekeying and/or bring the MTRH into a consistent state, the root user is warned and allowed to force mount. Otherwise, the system will not initialize

[0102] For subsequent rekeying efforts in the latter two cases, rather than incrementing the corresponding keystore counters by 1 during rekeying, they are incremented by 2. This is done to prevent potential reuse of any derived nugget keys that might have been in use right before the system crashed.

[0103] Thus, when the system can detect tampering, it will not initialize. When the system cannot distinguish between tampering and a crash, it offers the root user a choice to force mount. Thus, an attacker could force a crash and use root access to force mount. It is assumed, however, that if an attacker has root access to a device, its security is already compromised.

IV. Example Implementation

[0104] An example implementation of the embodiments described herein is comprised of 5000 lines of C code. Libraries used include OpenSSL version 1.0.2 and LibSodium version 1.0.12 for its ChaCha20, Argon2, Blake2, and AES-XTS implementations, likewise implemented in C. The SHA-256 Merkle Tree implementation is borrowed from the Secure Block Device library. To reduce the complexity of the experimental setup and allow execution in user space, a virtual device interface is provided through the BUSE virtual block device layer, itself based on the Network Block Device (NBD).

[0105] A. Deriving Subkeys

[0106] The cryptographic driver uses a shared master secret. The derivation of this master secret is implementation specific and has no impact on performance as it is completed during initialization. The implementation uses the Argon2 KDF to derive a master secret from a given password with an acceptable time-memory trade-off.

[0107] To assign each nugget its own unique keystream, each nugget uses a unique key and associated nonce. These nugget subkeys are derived from the master secret during initialization. To guarantee the backing store's integrity, each flake is tagged with a MAC. In this example implementation, the Polyl305 MAC is used, accepting a 32-byte one-time key and a plaintext of arbitrary length to generate tags. These one-time flake subkeys are derived from their respective nugget subkeys. In alternative embodiments, a hash-based message authentication code (HMAC), message authentication code based on universal hashing (UMAC), vhash-based message authentication code (VMAC), non-keyed hashing function (e.g., SHA2), or any other algorithm that can securely map a block of data to a unique tag could be used.

[0108] B. A Secure, Persistent, Monotonic Counter

[0109] The target platform uses an embedded Multi-Media Card (eMMC) as a backing store. In addition to boot and user data partitions, the eMMC standard includes a secure storage partition called a Replay Protected Memory Block (RPMB). The RPMB partition's size is configurable to be at most 16 megabytes (32 megabytes on some devices). All read and write commands issued to the RPMB are authenticated by a key burned into write-once storage (typically eFUSE) during a one-time, secure initialization process.

[0110] To implement rollback protection on top of the RPMB, the key for authenticating RPMB commands can be contained in TEE sealed storage or derived from the TPM. For this implementation, the system interacts with TPM/TEE secure storage only at mount time, where the authentication key can be retrieved and cached for the duration of the system's lifetime. With the cached key on hand, the implementation makes traditional IOCTL calls to read and write global version counter data to the RPMB eMMC partition, enforcing the invariant that it only increase monotonically.

[0111] The design is not dependent on the eMMC standard, however. Trusted hardware mechanisms other than the eMMC RPMB partition, including TPMs, support secure, persistent storage and/or monotonic counters directly. These can be adapted for use as well. Further, any interface that makes secure monotonic counters available can be used. For example, if a future operating system or hypervisor provided secure monotonic counters, that could be used instead.

[0112] There are two practical concerns to be addressed while implementing the secure counter: wear and performance overhead. Wear is a concern because the counter is implemented in non-volatile storage. The RPMB implements all the same wear protection mechanisms that are used to store user-data. Additionally, the system writes to the global version counter once per write to user-data. Given that the eMMC implements the same wear protection for the RPMB and user data, and that the ratio of writes to these areas is 1:1, it is expected that the system places no additional wear burden on the hardware. Further, with the JEDEC spec suggesting RPMB implementations use more durable and faster single-level NAND flash cells rather than cheaper and slower multi-level NAND flash cells, the RPMB partition will likely outlive and outperform the user-data portion of the eMMC.

[0113] In terms of performance overhead, updating the global version counter involves making one 64-bit authenticated write per user-data write. As user-data writes are almost always substantially larger, there is no significant overhead from the using the RPMB to store the secure counter.

[0114] C. LFS Garbage Collection

[0115] An LFS attempts to write to a drive sequentially in an append-only fashion, as if writing to a log. This requires large amounts of contiguous space, called segments. Since any backing store is necessarily finite, an LFS can only append so much data before it runs out of space. When this occurs, the LFS triggers a segment cleaning algorithm to erase outdated data and compress the remainder of the log into as few segments as possible. This procedure is known more broadly as garbage collection.

[0116] In the context of the embodiments herein, garbage collection could potentially incur high overhead. The procedure itself would, with its every write, require a rekeying of any affected nuggets. Worse, every proceeding write would appear to the system as if it were an overwrite, since there is no way for the system to know that the LFS triggered garbage collection internally.

[0117] In practice, modern production LFSes are optimized to perform garbage collection as few times as possible. Further, they often perform garbage collection in a background thread that triggers when the filesystem is idle and only perform expensive on-demand garbage collection when the backing store is nearing capacity. Garbage collection was turned on for all tests and there was no substantial performance degradation from this process because it is scheduled not to interfere with user I/O.

[0118] D. Overhead

[0119] The system stores metadata on the drive it is encrypting (see FIG. **5**). This metadata should be small compared to the user data. The implementation uses 4-kilobyte flakes, 256 flakes/nugget, and 1024 nuggets per gigabytes of user data. Given the flake and nugget overhead, this configuration requires just over 40 kilobytes of metadata per 1 gigabyte of user data. There is an additional, single static header that requires just over 200 bytes. Thus, the system's overhead in terms of storage is less than one hundredth of a percent.

[0120] V. Experimental Evaluation

[0121] A. Setup

[0122] A prototype was implemented on a Hardkernel Odroid XU3 ARM big.LITTLE system (Samsung Exynos 5422 A15 and A7 quad core CPUs, 2 gigabytes of LPDDR3 RAM, eMMC5.0 HS400 backing store) running Ubuntu Trusty 14.04 LTS, kernel version 3.10.58.

[0123] B. Methodology

[0124] To evaluate the performance of the embodiments herein, the latency (seconds/milliseconds per operation) of both sequential and random read and write I/O operations across four different standard Linux filesystems was measured. These filesystems are NILFS2, F2FS, Ext4 in ordered journaling mode, and Ext4 in full journaling mode. The I/O operations were performed using file sizes between 4 kilobytes and 40 megabytes. These files were populated with random data. The experiments were performed using a standard Linux ramdisk (tmpfs) as the ultimate backing store.

[0125] Ext4's default mode is ordered journaling mode (data=ordered), where metadata is committed to the filesystem's journal while the actual data is written through to the main filesystem. In the case of a crash, the filesystem uses the journal to avoid damage and recover to a consistent state. Full journaling mode (data journal) journals both metadata and the filesystem's actual data—essentially a double write-back for each write operation. In the case of a crash, the journal can replay entire I/O events so that both the filesystem and its data can be recovered. Both modes of Ext4 were considered to further explore the impact of frequent overwrites.

[0126] The experiment consists of reading and writing each file in its entirety 30 times sequentially, and then reading and writing random portions of each file 30 times. In both cases, the same amount of data is read and written per file. The median latency is taken per result set. The choice of 30 read/write operations (10 read/write operations repeated three times each) was to handle potential variation. The Linux page cache is dropped before every read operation, each file is opened in synchronous I/O mode via O_SYNC, and non-buffered read( )/write( ) system calls were used. A high-level I/O size of 128 kilobytes was used for all read and write calls that impact the filesystems; however, the I/O requests being made at the block device layer varied between 4 kilobytes and 128 kilobytes depending on the filesystem under test.

[0127] The experiment was repeated on each filesystem in three different configurations. The first configuration is unencrypted. The filesystem is mounted atop a BUSE virtual block device set up to immediately pass through any incoming I/O requests straight to the backing store. This is the baseline measurement of the filesystem's performance without any encryption. The second configuration uses the embodiments herein. The filesystem is mounted atop a BUSE virtual block device, provided by the implementation described above, to perform full-disk encryption. The third configuration uses dm-crypt. The filesystem is mounted atop a Device Mapper higher-level virtual block device provided by dm-crypt to perform full-disk encryption, which itself is mounted atop a BUSE virtual block device with pass through behavior identical to the device used in the baseline configuration. The dm-crypt module was configured to use AES-XTS as its full-disk encryption algorithm. All other parameters were left at their default values.

[0128] FIGS. **7**A and **7**B compare the embodiments herein to dm-crypt under the F2FS filesystem. The gamut of result sets over different filesystems can be seen in FIG. **8**A-**8**D. FIGS. **9**A and **9**B compare Ext4 with dm-crypt to F2FS with the embodiments herein. In these figures, an implementation of the embodiments herein is referred to as "StrongBox". However, other implementations are possible.

[0129] C. Read Performance

[0130] FIGS. **7**A and **7**B show the read performance of the embodiments herein in comparison to dm-crypt, both mounted with the F2FS filesystem. The disclosed embodiments improve on the performance of dm-crypt's AES-XTS implementation across sequential and random read operations on all file sizes. Specifically, the improvements are 2.07× for sequential 40-megabyte reads, 2.08× for sequential 5-megabyte reads, 1.85× for sequential 512-kilobyte reads, and 1.03× for sequential 4-kilobyte reads.

[0131] FIGS. **8**A and **8**C provide an expanded performance profile, testing a gamut of filesystems broken down by workload file size. For sequential reads across all filesystems and file sizes, the implementations herein outperform dm-crypt. This is true even on the non-LFS Ext4 filesystems. Specifically, read performance improvements over dm-crypt AES-XTS for 40-megabyte sequential reads are 2.02x for NILFS, 2.07x for F2FS, 2.09x for Ext4 in ordered journaling mode, and 2.06× for Ext4 in full journaling mode. For smaller file sizes, the performance improvement is less pronounced. For 4-kilobyte reads, the improvements are 1.28× for NILFS, 1.03× for F2FS, 1.07× for Ext4 in ordered journaling mode, and 1.04× for Ext4 in full journaling mode. When it comes to random reads, there are virtually identical results save for 4-kilobyte reads, where dm-crypt proved very slightly more performant under the NILFS LFS at 1.12×. This behavior is not observed with the more modern F2FS.

[0132] D. Write Performance

[0133] FIGS. **7**A and **7**B show the performance of the embodiments herein in comparison to dm-crypt under the modern F2FS LFS broken down by workload file size. Similar to read performance under the F2FS, these embodiments improve on the performance of dm-crypt's AES-XTS implementation across sequential and random write operations on all file sizes. Hence, the embodiments herein under F2FS are holistically faster than dm-crypt under F2FS. Specifically, the improvements are 1.33× for sequential 40-megabyte writes, 1.21× for sequential 5-megabyte writes, 1.15× for sequential 512-kilobyte writes, and 1.19× for sequential 4-kilobyte writes.

[0134] FIGS. **8**B and **8**D show an expanded performance profile, testing a gamut of filesystems broken down by workload file size. Unlike read performance, write performance under certain filesystems shows some improvements but not for all tests. For 40-megabyte sequential writes, the embodiments herein outperform dm-crypt's AES-XTS implementation by 1.33× for F2FS and 1.18× for NILFS. When it comes to Ext4, write performance drops, with a 3.6× slowdown for both ordered journaling and full journaling modes. For non-LFS 4-kilobyte writes, the performance degradation is even more pronounced with a 8.09× slowdown for ordered journaling and 14.5× slowdown for full journaling.

[0135] This slowdown occurs in Ext4 because, while writes from non-LFS filesystems have a metadata overhead that is comparable to that of forward writes in an LFS filesystem, Ext4 is not an append-only or append-mostly filesystem. This means that, at any time, Ext4 will initiate one or more overwrites anywhere on the disk (see Table 1). As described above, overwrites, once detected, trigger the rekeying process, which is a relatively expensive operation. Multiple overwrites compound this expense further. This makes Ext4 and other filesystems that do not exhibit at least append-mostly behavior likely unsuitable for use with the embodiments herein.

[0136] For both sequential and random 4-kilobyte writes among the LFSs, the performance improvement over dm-crypt's AES-XTS implementation for LFSs deflates. For the more modern F2FS atop the embodiments herein, there is a 1.19× improvement. For the older NILFS filesystem atop the embodiments herein, there is a 2.38× slowdown. This is where the overhead associated with tracking writes and detecting overwrites potentially becoming problematic, though the overhead is negligible depending on choice of LFS and workload characteristics.

[0137] These results show that the embodiments herein are sensitive to the behavior of the LFS that is mounted atop it, and that any practical use would require an extra profiling step to determine which LFS works best with a specific workload. With the correct selection of LFS, such as F2FS for workloads dominated by small write operations, potential slowdowns when compared to mounting that same filesystem over dm-crypt's AES-XTS can be effectively mitigated.

[0138] E. Replacing dm-crypt and Ext4

[0139] FIGS. **9**A and **9**B show the performance benefit of using the embodiments herein with F2FS over the popular dm-crypt with Ext4 in ordered journaling mode combination for both sequential and random read and write operations of various sizes. Other than 4-kilobyte write operations, which is an instance where baseline F2FS without modification is simply slower than baseline Ext4 without dm-crypt, the embodiments herein with F2FS outperforms dm-crypt's AES-XTS implementation with Ext4.

[0140] These results show that configurations taking advantage of the popular combination of dm-crypt, AES-XTS, and Ext4 could see a significant improvement in read performance without a degradation in write performance except in cases where small (>512 kilobyte) writes dominate the workload.

[0141] Note, however, that several implicit assumptions exist in the above design. For one, it is presumed that there is ample memory at hand to house the Merkle Tree and all other data abstractions. Efficient memory use was not a goal of the implementation. In an implementation aiming to be production ready, much more memory efficient data structures would be utilized.

[0142] It is also for this reason that populating the Merkle Tree necessitates a rather lengthy mounting process. In tests, a 1-gigabyte backing store on the Odroid system can take as long as 15 seconds to mount.

[0143] F. ChaCha20 vs. AES Performance

[0144] FIGS. **7**A-**8**D give strong evidence for general performance improvement over dm-crypt not being an artifact of filesystem choice. Excluding Ext4 as a non-LFS filesystem, tests show that the embodiments herein outperform dm-crypt under an LFS filesystem in the vast majority of outcomes.

[0145] FIG. **10** depicts the relationship between ChaCha20, the stream cipher used in the tested implementation, and the AES cipher. The dm-crypt module implements AES in XTS mode to provide full-disk encryption functionality. Swapping out ChaCha20 for AES-CTR (AES in CTR mode makes AES act as if it was a stream cipher) resulted in slowdowns of up to 1.33× for reads and 1.15× for writes across all configurations, as shown in FIG. **10**.

[0146] Finally, tests were carried out to determine whether the general performance improvement can be attributed to the implementation of the embodiments herein rather than the choice of stream cipher. This was tested by implementing AES in XTS mode on top of the embodiments herein using OpenSSL EVP. This use of OpenSSL AES-XTS experiences slowdowns of up to 1.6× for reads and 1.23× for writes across all configurations compared to using ChaCha20. Interestingly, while significantly less performant, this slowdown is not entirely egregious, and suggests that perhaps there are parts of the dm-crypt code base that would benefit from further optimization.

[0147] G. Threat Analysis

[0148] Table 3 lists possible attacks and their results. It can be inferred from these results and the design described herein that the threat model is addressed and confidentiality and integrity guarantees are maintained.

TABLE 3

Attack	Result	Explanation
Nugget user data in backing store is mutated out-of-band online.	The system immediately fails with exception on successive I/O request.	The MTRH is inconsistent.
Header metadata in backing store is mutated out-of-band online, making the MTRH inconsistent.	The system immediately fails with exception on successive I/O request.	The MTRH is inconsistent.
Backing store is rolled back to a previously consistent state while online.	The system immediately fails with exception on successive I/O request.	TPMGLOBALVER and RPMB secure counter out of sync.
Backing store is rolled back to a previously consistent state while offline, RPMB secure counter wildly out of sync.	The system refuses to mount; allows for force mount with root access.	TPMGLOBALVER and RPMB secure counter out of sync.
MTRH made inconsistent by mutating backing store out-of-band while offline, RPMB secure counter in sync.	The system refuses to mount.	TPMGLOBALVER and RPMB secure counter are in sync, yet illegal data manipulation occurred.

[0149] H. Improvements Summarized

[0150] The conventional wisdom is that securing data at rest requires that one must pay the high performance overhead of encryption with AES is XTS mode. The embodiments herein demonstrate that technological trends overturn this conventional wisdom: log-structured file systems and hardware support for secure counters make it practical to use a stream cipher to secure data at rest. In particular, an implementation which uses the ChaCha20 stream cipher and the Poly1305 MAC to provide secure storage can be used as a drop-in replacement for dm-crypt. Empirical results show that under F2FS, a modern, industrial-strength log-structured file system, the embodiments herein provide upwards of 2× improvement on read performance and 1.21× improvement on write performance. In fact, these results show such a system provides a higher performance replacement for Ext4 backed with dm-crypt.

VI. Example Operations

[0151] FIGS. **11**A and **11**B depict a flow chart illustrating an example embodiment. The process illustrated by FIGS. **11**A and **11**B may be carried out by a computing device, such as computing device **100**, and/or a cluster of computing devices, such as server cluster **200**. However, the process can be carried out by other types of devices or device subsystems. For example, the process could be carried out by a portable computer, such as a laptop or a tablet device, or a smartphone.

[0152] The embodiments of FIGS. **11**A and **11**B may be simplified by the removal of any one or more of the features shown therein. Further, these embodiments may be combined with features, aspects, and/or implementations of any of the previous figures or otherwise described herein.

[0153] In this section, a nugget is referred to as a "logical block" and a flake is referred to as a "sub-block." This terminology is used for clarity and precision.

[0154] In FIG. **11**A, step **1100** involves receiving a request to write data to a memory unit. The memory unit may be divided into one or more logical blocks, each of the logical blocks subdivided into groups of sub-blocks. Each of the logical blocks maps to one or more physical sectors of the memory unit. Any of the sub-blocks being used to store information are encrypted in accordance with a stream cipher. The memory unit maintains a transaction journal that marks each sub-block as either dirty or clean. The memory unit stores keycount values for each of the logical blocks.

[0155] Step **1102** involves determining that the request seeks to write a portion of the data to a particular sub-block of the groups of sub-blocks.

[0156] Step **1104** involves determining that the particular sub-block is marked as dirty in the transaction journal.

[0157] Step **1106** involves reading a particular logical block containing the particular sub-block from the memory unit.

[0158] Step **1108** involves decrypting the particular logical block in accordance with the stream cipher.

[0159] Step **1110** involves writing the portion of the data to the particular sub-block.

[0160] Turning to FIG. **11**B, step **1112** involves incrementing the keycount value associated with the particular logical block.

[0161] Step **1114** involves generating a key for the particular logical block in accordance with the stream cipher.

[0162] Step **1116** involves encrypting the particular logical block using the stream cipher, the key, and the keycount value as incremented.

[0163] Step **1118** involves writing the particular logical block as encrypted to the memory unit.

[0164] Some embodiments may involve further steps of: receiving a second request to write second data to the memory unit, determining that the second request seeks to write a portion of the second data to a second particular sub-block of the groups of sub-blocks, determining that the second particular sub-block is not marked as dirty in the transaction journal, marking the second particular sub-block as dirty in the transaction journal, reading a second particular logical block containing the second particular sub-block from the memory unit, decrypting the second particular logical block in accordance with the stream cipher, writing the portion of the second data to the second particular sub-block, generating a second key for the second particular logical block in accordance with the stream cipher, encrypting the second particular logical block using the stream cipher, second key, and a second keycount value associated with the second particular logical block, and writing the second particular logical block as encrypted to the memory unit.

[0165] In some embodiments, a hash tree contains hash outputs of each of the sub-blocks, and further steps may involve: after reading the particular logical block from the memory unit, validating the particular sub-block with the hash output associated with the particular sub-block, and after writing the particular logical block as encrypted to the memory unit, calculating a new hash output for the particular sub-block and updating the hash tree to associate the new hash output with the particular sub-block. The hash tree may be stored in the memory unit. The hash tree may be a Merkle Tree.

[0166] In some embodiments, the memory unit also maintains a rekeying journal that temporarily stores the particular logical block as decrypted until the particular logical block as encrypted is written to the memory unit.

[0167] Some embodiments may also include a replay protected memory block (RPMB) that stores a persistent monotonic counter, and further steps may involve: after writing the particular logical block as encrypted to the memory unit, updating the persistent monotonic counter in the RPMB, and storing a copy of the persistent monotonic counter in the memory unit. In some embodiments, the system may not initialize for non-privileged users if the persistent monotonic counter in the RPMB is not identical to the copy of the persistent monotonic counter in the memory unit.

[0168] Some embodiments may also include an operating system configured to access the memory unit by way of the cryptographic software module. The operating system may overlay a log-structured file system (or other types of file systems) atop of the memory unit. The file system may be based on F2FS.

[0169] Some embodiments may involve a master secret, and generating the key for the particular logical block in accordance with the stream cipher may involve generating the key based in part on the master secret. The stream cipher may be based on ChaCha20, for example.

VII. Additional Embodiments and Performance Results

[0170] The additional embodiments described in this section can be combined with any one or more of the previously-described embodiments.

[0171] In the embodiments above, it was shown that recent developments in mobile hardware invalidate the assumption that stream ciphers are unsuitable for FDE. Thus, fast stream ciphers can be used to dramatically improve the performance of FDE. In particular, modern mobile devices employ solid-state storage with FTL, which operate similarly to an LFS. They also include trusted hardware such as TEEs and secure storage areas. Embodiments using the ChaCha20 stream cipher leveraged these two trends to outperform dm-crypt, the de-facto Linux FDE endpoint.

[0172] In this section, embodiments using stream ciphers beyond ChaCha20 and AES-CTR are explored. Specifically, the following eSTREAM profile 1 stream ciphers (suitable for software applications with high throughput requirements) were considered: Sosemanuk, Rabbit, Salsa20, Salsa12, and Salsa8. ChaCha8/12 are not considered eSTREAM ciphers and so were not included in this comparison (but were included in later experimental implementations). Further, eSTREAM profile 2 stream ciphers were not explicitly considered but could potentially produce improved performance as well. In various embodiments, other stream ciphers or block ciphers with stream-cipher-like characteristics may be used.

[0173] A. Experimental Setup

[0174] Experiments were performed on a Hardkernel Odroid XU3 ARM big.LITTLE system (Samsung Exynos5422 A15 and A7 quad core CPUs, 2Gbyte LPDDR3 RAM, eMMC5.0 HS400 backing store) running Ubuntu Trusty 14.04 LTS, kernel version 3.10.58. To evaluate performance under these new ciphers, measurements included the latency (time per operation) of sequential read and write I/O operations against the F2FS LFS. The I/O operations were performed using 1 KiB and 5 MiB file sizes (where 1 KiB=2^{10} bytes and 1 MiB=2^{20} bytes). These files were populated with random data. The experiments are conducted using a standard Linux ramdisk (tmpfs) as the ultimate backing store. The I/O size used was the maximum that the Odroid XU3 kernel version 3.10.58 supports, selected by the operating system automatically.

[0175] B. Evaluation

[0176] For each of the figures in this section, the metrics indicate averages (medians) of 10 runs over the whole size of the file. Note that, except in the case of Salsa12, in figures where energy/power data is missing is due to the coarse timing resolution of current energy monitoring tools. Unfortunately, these specific tools cannot be easily made to operate at a faster frequency.

[0177] FIG. **12**A shows the performance of dm-crypt for 1 KiB whole file reads in comparison to the previous embodiments herein (which are again referred to interchangeably as "StrongBox") implemented with several stream ciphers, including use of the original ChaCha20 stream cipher. ChaCha20 is on average 1.09× faster than dm-crypt for reads at this file size, which is congruous with the results above. Sosemanuk is the worst performer, being 1.15× slower than ChaCha20. ChaCha20 is 1.01× faster than Salsa20 but 1.01× slower than Salsa8 and virtually maintains latency parity with Salsa 12. ChaCha20 is 1.05× faster than Rabbit.

[0178] FIG. **12**B shows the performance of dm-crypt for 1 KiB whole file writes in comparison to the embodiments herein implemented with several stream ciphers. On average, dm-crypt is 1.41 × faster than ChaCha20 for writes at this file size. Still, ChaCha20 is 1.05× faster than Rabbit, and 1.36× faster than Sosemanuk. Salsa20 is slightly slower than ChaCha20 at 1.01×. Salsa12 is 1.07× faster than ChaCha20. Salsa8 is 1.07× faster than ChaCha20.

[0179] Similar to FIG. **12**A, FIG. **12**C shows the performance of dm-crypt against the embodiments herein, but for 5 MiB whole file reads. ChaCha20 is 1.95× faster than dm-crypt for reads at this file size, which is congruous with the results above. The only cipher faster than ChaCha20 at this size is Salsa8, which is 1.03× faster than ChaCha20. Sosemanuk virtually maintains latency parity with dm-crypt in that it is 1.95× slower than ChaCha20 at this size. ChaCha20 is 1.04× faster than Salsa12. ChaCha20 is 1.18× faster than Salsa20. ChaCha20 is 1.3 8× faster than Rabbit.

[0180] Similar to FIG. **12**B, FIG. **12**D shows the performance of dm-crypt against the embodiments herein for 5 MiB whole file writes. Here, ChaCha20 is 1.02× faster than dm-crypt for writes at this file size. ChaCha20 is 1.10× faster than Rabbit. ChaCha20 is 1.25× faster than Sosemanuk. ChaCha20 is 1.04× faster than Salsa20. Salsa12 is 1.02× faster than ChaCha20, while Salsa8 is 1.04× faster than ChaCha20.

[0181] Evident from the above is the fact that writes have slowed down for StrongBox based implementations. This slowdown is most likely due to the new software layer used to facilitate cipher switching that was added for the measurements of FIGS. **12A-12**D. Fortunately, the slowdown only seems to affect writes. The performance win with ChaCha20 reads over dm-crypt remains nearly two-to-one. One solution that immediately presents itself is to improve these embodiments by offloading I/O operations to an (un) bounded thread pool, which is a distinct advantage the production-ready dm-crypt software current employs.

[0182] The SalsaX (Salsa20, Salsa12, Salsa8) functions outperforming ChaCha20 by a small factor in most instances above comes with a heavy caveat: increased or otherwise unusual energy/power use. In several cases, such as with FIG. **12**C, the performance win might be entirely outweighed by the efficiency loss, depending on the use case scenario. This was not entirely unexpected, since the reasons for the ChaChaX family of alternative implementations being created in the first place included increased energy efficiency. The ramifications of cipher selection on total system energy use are of paramount concern in many practical scenarios.

VIII. Conclusion

[0183] The present disclosure is not to be limited in terms of the particular embodiments described in this application, which are intended as illustrations of various aspects. Many modifications and variations can be made without departing from its scope, as will be apparent to those skilled in the art. Functionally equivalent methods and apparatuses within the scope of the disclosure, in addition to those described herein, will be apparent to those skilled in the art from the foregoing descriptions. Such modifications and variations are intended to fall within the scope of the appended claims.

[0184] The above detailed description describes various features and operations of the disclosed systems, devices, and methods with reference to the accompanying figures. The example embodiments described herein and in the figures are not meant to be limiting. Other embodiments can be utilized, and other changes can be made, without departing from the scope of the subject matter presented herein. It will be readily understood that the aspects of the present disclosure, as generally described herein, and illustrated in the figures, can be arranged, substituted, combined, separated, and designed in a wide variety of different configurations.

[0185] With respect to any or all of the message flow diagrams, scenarios, and flow charts in the figures and as discussed herein, each step, block, and/or communication can represent a processing of information and/or a transmission of information in accordance with example embodiments. Alternative embodiments are included within the scope of these example embodiments. In these alternative embodiments, for example, operations described as steps, blocks, transmissions, communications, requests, responses, and/or messages can be executed out of order from that shown or discussed, including substantially concurrently or in reverse order, depending on the functionality involved. Further, more or fewer blocks and/or operations can be used with any of the message flow diagrams, scenarios, and flow charts discussed herein, and these message flow diagrams, scenarios, and flow charts can be combined with one another, in part or in whole.

[0186] A step or block that represents a processing of information can correspond to circuitry that can be configured to perform the specific logical functions of a herein-described method or technique. Alternatively or additionally, a step or block that represents a processing of information can correspond to a module, a segment, or a portion of program code (including related data). The program code can include one or more instructions executable by a processor for implementing specific logical operations or actions in the method or technique. The program code and/or related data can be stored on any type of computer readable medium such as a storage device including RAM, a disk drive, a solid state drive, or another storage medium.

[0187] The computer readable medium can also include non-transitory computer readable media such as computer readable media that store data for short periods of time like register memory and processor cache. The computer readable media can further include non-transitory computer readable media that store program code and/or data for longer periods of time. Thus, the computer readable media may include secondary or persistent long term storage, like ROM, optical or magnetic disks, solid state drives, compact-disc read only memory (CD-ROM), for example. The computer readable media can also be any other volatile or non-volatile storage systems. A computer readable medium can be considered a computer readable storage medium, for example, or a tangible storage device.

[0188] Moreover, a step or block that represents one or more information transmissions can correspond to information transmissions between software and/or hardware modules in the same physical device. However, other information transmissions can be between software modules and/or hardware modules in different physical devices.

[0189] The particular arrangements shown in the figures should not be viewed as limiting. It should be understood that other embodiments can include more or less of each element shown in a given figure. Further, some of the illustrated elements can be combined or omitted. Yet further, an example embodiment can include elements that are not illustrated in the figures.

[0190] While various aspects and embodiments have been disclosed herein, other aspects and embodiments will be apparent to those skilled in the art. The various aspects and embodiments disclosed herein are for purpose of illustration and are not intended to be limiting, with the true scope being indicated by the following claims.

What is claimed is:

1. A system comprising:

a memory unit divided into one or more logical blocks, each of the logical blocks subdivided into groups of sub-blocks, wherein each of the logical blocks maps to one or more physical sectors of the memory unit, wherein any of the sub-blocks being used to store information are encrypted in accordance with a stream cipher, wherein the memory unit maintains a transaction journal that marks each sub-block as either dirty or clean, and wherein the memory unit stores keycount values for each of the logical blocks; and

a cryptography software module, configured to perform operations comprising:

receiving a request to write data to the memory unit,

determining that the request seeks to write a portion of the data to a particular sub-block of the groups of sub-blocks,

determining that the particular sub-block is marked as dirty in the transaction journal,

reading a particular logical block containing the particular sub-block from the memory unit,

decrypting the particular logical block in accordance with the stream cipher,

writing the portion of the data to the particular sub-block,

incrementing the keycount value associated with the particular logical block,

generating a key for the particular logical block in accordance with the stream cipher,

encrypting the particular logical block using the stream cipher, the key, and the keycount value as incremented, and

writing the particular logical block as encrypted to the memory unit.

2. The system of claim **1**, wherein the cryptography software module is further configured to perform operations comprising:

receiving a second request to write second data to the memory unit;

determining that the second request seeks to write a portion of the second data to a second particular sub-block of the groups of sub-blocks;

determining that the second particular sub-block is not marked as dirty in the transaction journal;

marking the second particular sub-block as dirty in the transaction journal;

reading a second particular logical block containing the second particular sub-block from the memory unit;

decrypting the second particular logical block in accordance with the stream cipher;

writing the portion of the second data to the second particular sub-block;

generating a second key for the second particular logical block in accordance with the stream cipher;

encrypting the second particular logical block using the stream cipher, second key, and a second keycount value associated with the second particular logical block; and

writing the second particular logical block as encrypted to the memory unit.

3. The system of claim **1**, wherein the cryptography software module has access to a hash tree that contains hash outputs of each of the sub-blocks, and wherein the cryptography software module is further configured to perform operations comprising:

after reading the particular logical block from the memory unit, validating the particular sub-block with the hash output associated with the particular sub-block; and

after writing the particular logical block as encrypted to the memory unit, calculating a new hash output for the particular sub-block and updating the hash tree to associate the new hash output with the particular sub-block.

4. The system of claim **3**, wherein the hash tree is stored in the memory unit.

5. The system of claim **3**, wherein the hash tree is a Merkle Tree.

6. The system of claim **1**, wherein the memory unit also maintains a rekeying journal that temporarily stores the particular logical block as decrypted until the particular logical block as encrypted is written to the memory unit.

7. The system of claim **1**, further comprising a replay protected memory block (RPMB) that stores a persistent monotonic counter, wherein the cryptography software module is further configured to perform operations comprising:

after writing the particular logical block as encrypted to the memory unit, updating the persistent monotonic counter in the RPMB; and

storing a copy of the persistent monotonic counter in the memory unit.

8. The system of claim **7**, wherein the cryptography software module will not initialize for non-privileged users if the persistent monotonic counter in the RPMB is not identical to the copy of the persistent monotonic counter in the memory unit.

9. The system of claim **1**, further comprising:

an operating system configured to access the memory unit by way of the cryptographic software module.

10. The system of claim **9**, wherein the operating system overlays a log-structured file system atop of the memory unit.

11. The system of claim **10**, wherein the log-structured file system is based on F2FS.

12. The system of claim **1**, wherein the cryptographic software module has access to a master secret, and wherein generating the key for the particular logical block in accordance with the stream cipher comprises generating the key based in part on the master secret.

13. The system of claim **1**, wherein the stream cipher is based on ChaCha20.

14. A computer-implemented method comprising:

receiving a request to write data to a memory unit, wherein the memory unit is divided into one or more logical blocks, each of the logical blocks subdivided into groups of sub-blocks, wherein each of the logical blocks maps to one or more physical sectors of the memory unit, wherein any of the sub-blocks being used to store information are encrypted in accordance with a stream cipher, wherein the memory unit maintains a transaction journal that marks each sub-block as either dirty or clean, and wherein the memory unit stores keycount values for each of the logical blocks;

determining that the request seeks to write a portion of the data to a particular sub-block of the groups of sub-blocks;

determining that the particular sub-block is marked as dirty in the transaction journal;

reading a particular logical block containing the particular sub-block from the memory unit;

decrypting the particular logical block in accordance with the stream cipher;

writing the portion of the data to the particular sub-block;

incrementing the keycount value associated with the particular logical block;

generating a key for the particular logical block in accordance with the stream cipher;

encrypting the particular logical block using the stream cipher, the key, and the keycount value as incremented; and

writing the particular logical block as encrypted to the memory unit.

15. The computer-implemented method of claim **14**, further comprising:

receiving a second request to write second data to the memory unit;

determining that the second request seeks to write a portion of the second data to a second particular sub-block of the groups of sub-blocks;

determining that the second particular sub-block is not marked as dirty in the transaction journal;

marking the second particular sub-block as dirty in the transaction journal;

reading a second particular logical block containing the second particular sub-block from the memory unit;

decrypting the second particular logical block in accordance with the stream cipher;

writing the portion of the second data to the second particular sub-block;

generating a second key for the second particular logical block in accordance with the stream cipher;

encrypting the second particular logical block using the stream cipher, second key, and a second keycount value associated with the second particular logical block; and

writing the second particular logical block as encrypted to the memory unit.

16. The computer-implemented method of claim **14**, wherein a hash tree contains hash outputs of each of the sub-blocks, the method further comprising:

after reading the particular logical block from the memory unit, validating the particular sub-block with the hash output associated with the particular sub-block; and

after writing the particular logical block as encrypted to the memory unit, calculating a new hash output for the particular sub-block and updating the hash tree to associate the new hash output with the particular sub-block.

17. The computer-implemented method of claim **14**, wherein the memory unit also maintains a rekeying journal that temporarily stores the particular logical block as decrypted until the particular logical block as encrypted is written to the memory unit.

18. The computer-implemented method of claim **14**, wherein a replay protected memory block (RPMB) stores a persistent monotonic counter, the method further comprising:

after writing the particular logical block as encrypted to the memory unit, updating the persistent monotonic counter in the RPMB; and

storing a copy of the persistent monotonic counter in the memory unit.

19. The computer-implemented method of claim **18**, wherein a cryptography software module performs all reads to and writes from the memory unit, and wherein the cryptography software module will not initialize for non-privileged users if the persistent monotonic counter in the RPMB is not identical to the copy of the persistent monotonic counter in the memory unit.

20. An article of manufacture including a non-transitory computer-readable medium, having stored thereon program instructions that, upon execution by a computing system, cause the computing system to perform operations comprising:

receiving a request to write data to a memory unit of the computing system, wherein the memory unit is divided into one or more logical blocks, each of the logical blocks subdivided into groups of sub-blocks, wherein each of the logical blocks maps to one or more physical sectors of the memory unit, wherein any of the sub-blocks being used to store information are encrypted in accordance with a stream cipher, wherein the memory unit maintains a transaction journal that marks each sub-block as either dirty or clean, and wherein the memory unit stores keycount values for each of the logical blocks;

determining that the request seeks to write a portion of the data to a particular sub-block of the groups of sub-blocks;

determining that the particular sub-block is marked as dirty in the transaction journal;

reading a particular logical block containing the particular sub-block from the memory unit;

decrypting the particular logical block in accordance with the stream cipher;

writing the portion of the data to the particular sub-block;

incrementing the keycount value associated with the particular logical block;

generating a key for the particular logical block in accordance with the stream cipher;

encrypting the particular logical block using the stream cipher, the key, and the keycount value as incremented; and

writing the particular logical block as encrypted to the memory unit.

* * * * *