Forward Security with Crash Recovery for Secure Logs

ERIK-OLIVER BLASS, Airbus, Germany
GUEVARA NOUBIR, Northeastern University, USA

Logging is a key mechanism in the security of computer systems. Beyond supporting important forward security properties, it is critical that logging withstands both failures and intentional tampering to prevent subtle attacks leaving the system in an inconsistent state with inconclusive evidence. We propose new techniques combining forward security with crash recovery for secure log data storage. As the support of specifically forward integrity and the online nature of logging prevent the use of conventional coding, we propose and analyze a coding scheme resolving these unique design constraints. Specifically, our coding enables forward integrity, online encoding, and most importantly a constant number of operations per encoding. It adds a new log item by XORing it to k cells of a table. If up to a certain threshold of cells is modified by the adversary, or lost due to a crash, we still guarantee recovery of all stored log items. The main advantage of the coding scheme is its efficiency and compatibility with forward integrity. The key contribution of the paper is the use of spectral graph theory techniques to prove that k is constant in the number n of all log items ever stored and small in practice, e.g., $k = 5$. Moreover, we prove that to cope with up to $\sqrt{n}$ modified or lost log items, storage expansion is constant in n and small in practice. For $k = 5$, the size of the table is only 12% more than the simple concatenation of all n items. We propose and evaluate original techniques to scale the computation cost of recovery to several GBytes of security logs. We instantiate our scheme into an abstract data structure which allows to either detect adversarial modifications to log items or treat modifications like data loss in a system crash. The data structure can recover lost log items, thereby effectively reverting adversarial modifications.

1 INTRODUCTION

Log services such as `Syslog` collect data about security-relevant events. Logged data is important for security audits and used during forensic analysis, where an analyst investigates how an adversary has attacked a system. Yet, if the adversary is able to fully compromise the machine running the log service, they typically modify stored log data and remove all traces of their attack. As a result, an analyst checking for attacks does not have any means to verify integrity and truthfulness of logs. To cope with compromising adversaries, previous works have designed mechanisms to store logs with *forward integrity*, see [3–6, 8, 18–20, 27–29, 31, 33, 37, 46, 47].

An adversary who has compromised a system has full read-write access to all system data and can therefore easily modify previously logged data items. Forward integrity ensures that a data item logged at time t is integrity protected such that an adversary compromising the system at time $t' > t$ cannot modify it without being detected. Consequently, the goal of forward integrity is not to prevent modifications to data, but to make modifications evident ("tamper evidence") during log analysis. The standard notion of forward integrity is rather simple to achieve. In addition to storing the i^{th} log item $data_i$ in a log file, the log service also stores $\mathsf{HMAC}_{K_i}(data_i)$. Key K_i is evolved to K_{i+1} by computing $K_{i+1} = \mathsf{PRF}_{K_i}(0)$, and K_i is discarded. An adversary compromising the system at time $i + 1$ learns K_{i+1}, but not K_i and thus cannot modify previous HMACs without detection. The log service starts with key K_0 which is also known to the analyst.

Yet, a real-world challenge arises from the problem that log files can become inconsistent. Systems crash for various reasons like software bugs, power failure or even hardware failure. As a result, log data might only be partially written to disk, or previously written data or integrity information such as the HMACs become corrupted. In case of a crash, the analyst would need to accept bad integrity information as a potential crash inconsistency. As demonstrated before [3, 6], adversaries can exploit crashes by performing crash attacks: after compromise, an adversary removes or modifies traces

Authors' addresses: Erik-Oliver Blass, erik-oliver.blass@airbus.com, Airbus, Munich, Germany; Guevara Noubir, g.noubir@northeastern.edu, Northeastern University, Boston, MA, USA.

of their attack and crashes the system. Again, the analyst would accept inconsistent integrity information as a result of the crash, allowing the adversary to evade detection.

In many logging scenarios where security-critical data is collected, not only the integrity but also the confidentiality of data must be protected. An adversary who has compromised the system should not be able to get access to sensitive data collected before the time of compromise, e.g., about other users' logins and accesses to files and other resources. That is, we want forward confidentiality. Together, forward confidentiality and forward integrity are called forward security.

This paper. To mitigate the above attacks, we propose techniques combining forward integrity and confidentiality with crash recovery for secure data storage. In particular, we present a new coding scheme with unique design constraints such as forward integrity and most importantly an *online encoding* with a *constant number* of operations per encoded symbol which is not possible with typical error/erasure correction codes such as LDPC, Reed-Solomon, or Fountain codes. In this context, *online encoding* refers to the property that a symbol is encoded at once and without knowledge of previously encoded symbols.

While the proposed scheme is a random linear code, it has not been considered or analyzed before, as it addresses special constraints of secure logging. It is not suitable for typical communications and storage scenarios. We instantiate the proposed coding techniques in a new abstract data structure Π with operations addItem and listItems. Operation addItem($data$) adds data item $data$ (a bit string) to Π, and listItems outputs all data items in the order they have been previously added. Data structure Π is useful for, e.g., storing a sequence of incoming log entries, but we stress that Π is general, and one can conceive other applications. Besides providing forward integrity and similarly forward confidentiality, the crucial feature of Π is the ability to recover data in case of system crashes with data loss or in case of data corruption.

Specifically, if up to some amount δ of Π's internal data representation gets deleted or corrupted, then listItems still recovers and outputs all data items with high probability. As δ is a parameter for Π, it is chosen such that it matches the expected amount of data lost during a real-world crash, e.g., due to the system's cache sizes, cache eviction frequency, and file system details. Consequently, the adversary can only modify up to δ data, otherwise malicious modifications become distinguishable from a real crash and lead to detection. Yet, if the adversary modifies at most δ data, listItems will recover all original data, neutralizing the adversary's modifications.

The key technical challenge in the design of such a data structure is to combine forward integrity and recovery, but still achieve high efficiency in terms of computational complexity and low storage overhead. In many logging scenarios, a log service must be able to cope with a high frequency of incoming log events.

Coding Overview. Our coding bears similarity to Gallager's Low-density Parity-Check (LDPC) codes [16, 40]. We, however, have unique constraints that prevent the use of such codes. In order to provide forward integrity and for performance reasons, the encoding has to happen in an online manner without knowledge of previous encodings, and an addItem operation must only imply a constant number of (simple) computations and disk writes. These requirements *exclude the use of conventional codes* such as LDPC codes [35], or even Reed-Solomon and Fountain Codes (e.g., LT, Tornado, and Raptor) as we discuss in the related work section. Our encoding forward-securely selects k pseudo-random locations in a table and XORs encrypted data to these locations. So, we build a system of linear equations, with the table cells representing its right-hand side and indices of pseudo-random locations its left-hand side. We show that if k and the size of the table are chosen appropriately, then the matrix of coefficients of the left-hand side has full rank with high probability $1 - o(1)$. This allows for decodability, i.e., we recover all data using standard Gaussian elimination. More importantly, even if an amount of up to δ data items in the table becomes invalid, and thus a certain number of equations are removed from the system, the left-hand side has still full rank, and we recover all data.

While the coding scheme is purposefully simple for high efficiency, our main contribution lies in its analysis where we show for which parameters decoding can be guaranteed with high probability. In order to prove decodability guarantees, we extend Calkin's analysis [10] of dependent sets of constant weight binary vectors to binary vectors of hypergeometrically distributed weight. The analysis is of independent interest and leverages spectral graph theoretic techniques: we connect a transition matrix' binary eigenvectors, which have hypergeometrically distributed weight, to the rank of such vectors. We also show tightness of decodability bounds.

To also achieve forward confidentiality, we combine our coding scheme with standard techniques for authenticated encryption. By using the same evolving keys as for integrity, we achieve confidentiality for "free".

In summary, the *major contributions of this paper* are:

- We present a new online encoding scheme for forward integrity, formally prove its security and decodability, and analyze its complexity. Our proof shows that encoding is extremely efficient and has $\Theta(k)$ time complexity, constant in the number n of data items encoded. Time complexity is not only asymptotically optimal, but also low in practice (e.g., $k = 5$). Computation time is dominated by k applications of cheap symmetric-key cryptography. Moreover, space overhead is in $\Theta(\frac{1}{1-e^{-k}})$ which is also constant in n and low in practice (12% for $k = 5$). Contrary to related work [17, 32], our coding scheme tolerates not only a constant amount δ of data loss, but any $\delta < \sqrt{n}$.
- Augmented by standard techniques for forward confidentiality, we deploy our coding scheme into data structure Π with operations addItem (performing encoding) and listItems (performing decoding). As a result, Π essentially allows ignoring an adversary $\mathcal{A}$ tampering with data. Operation listItems will still output all data items previously added with probability $1 - o(1)$. As we assume $\mathcal{A}$ to have fully compromised the computer system running the log service, $\mathcal{A}$ could also modify or remove more than δ of all content. However in that case, $\mathcal{A}$ is detected with probability 1. All data loss due to a regular system crash is recoverable with probability $1 - o(1)$.
- We propose and evaluate additional techniques (e.g., multi-bucket storage) to scale the computation cost of recovery to several GBytes of security logs. We use our implementation in millions of experiments and indicate that, as long as $\delta < \sqrt{n}$, listItems outputs all data items with high probability.

2 MODELS AND SECURITY DEFINITIONS

System Overview. We start by introducing the general system setup we consider in this paper. We envision three parties:

(1) The *logging device*. The logging device collects security-relevant data from different sources over time and stores and maintains this data in an internal data structure. The logging device is initialized with some cryptographic information K_0 that it shares with the analyst. In our setting, the logging devices is trusted until compromised by the adversary. A logging device might also *crash* which can result in some modifications to its internal data structure. Compared to the analyst, we assume the logging device to be a lightweight device, e.g., an Internet of Things (IoT) device, so the main computation burden of any logging scheme must lie on the analyst.

(2) The *adversary*. At some point in time after the initialization, an adversary compromises the logging device. The adversary learns all information stored on the device at the time of compromise and can fully control the device's behavior. The goal of the adversary is twofold. First, the adversary wants to learn about data stored before the time of compromise. Second, the adversary wants to tamper with the device's internal data structure and the data stored before the time of compromise. However, the adversary wants to remain covert, i.e., not be detected by the analyst.

(3) The *analyst* (also called verifier). The analyst is initialized with the same cryptographic information K_0 as the logging device. At some point in time, possibly after the device has been compromised by the adversary, the

Table 1. Notation used in this paper

Symbol	Description
c	Stretch factor, determines how much longer Table $\mathcal{T}$ is compared to maximum number of data items n
δ	Threshold number of deleted or modified data values in data structure Π (up to where decoding succeeds)
DS	Internal state of data structure Π, comprises table $\mathcal{T}$
k	Number of buckets in Table $\mathcal{T}$ where new data is XORed to
K_i	Cryptographic keys
ℓ	Length of data items
m	Number of cells in Table $\mathcal{T}$
n	Maximum number of data items to store
Π	Abstract data structure to store data
ϕ	Number of buckets in multi-bucket technique
s	Security parameter
$st_{\mathcal{A}}$	Adversary $\mathcal{A}$'s internal state
$\mathcal{T}$	Table storing encoded data items
$\mathcal{T}_i$	Contents of Table $\mathcal{T}$ at time i
Enc, Dec, PRF, PRG, h	Encryption, decryption, pseudo-random function, pseudo-random generator, hash function
$\Delta(DS, DS')$	Function computing the number of different data values between states DS and DS'
DRN($seed$)	Generate k distinct pseudo-random numbers using $seed$
$L(n)$	number of data values in Π

analyst is given the internal data structure of the device (in practice, they might simply download it from the log device). The goal of the analyst is to learn all original data collected by the logging device, at least from before the time of a potential compromise or before the time of a crash. If this is not possible due to adversarial tampering, the analyst wants to find out that there actually was adversarial tampering and not a crash. The analyst is always trusted in our scenario. Note that we do not assume any network connection between the logging device and the analyst during the device's regular collection of data.

In general, data structures with operations are also called Abstract Data Types (ADTs). However, whenever the separation is clear in this paper, we simply refer to Π as a data structure. To allow proper reasoning about security and later data recovery, we briefly formalize both (simple) storage data structures and the threat model.

2.1 Data Structures for Storage

A storage data structure $\Pi = (\mathsf{Init}, \mathsf{addItem}, \mathsf{listItems}, DS)$ comprises state DS and the following three algorithms.

(1) $(DS, sk) \leftarrow \mathsf{Init}(1^s, n)$: on input a security parameter s and the maximum number n of data items which will be stored, Init outputs an empty state $DS \in \{0,1\}^{\mathsf{poly}(s)}$. Moreover, Init also outputs auxiliary bit string $sk \in \{0,1\}^s$. In our specific instance of Π later, sk will be a secret cryptographic key, the start of a key chain to ensure forward integrity and confidentiality.

(2) $DS' \leftarrow \mathsf{addItem}(data, DS)$: on input bit string $data \in \{0,1\}^*$, this algorithm adds $data$ to the data structure given by state DS. It outputs an updated state DS'. addItem does not require auxiliary information sk. In practice, addItem is typically run by the logging device for each $data$ item.

(3) $(data_1, \ldots, data_\eta) \vee \perp \leftarrow \mathsf{listItems}(DS, sk, n)$: on input a data structure's state DS and auxiliary information sk, listItems *either* outputs a sequence of data $data_i$ *or* special symbol $\perp$ indicating failure. To be able to output failure, e.g., in case of a crash, listItems also receives system parameter n. In practice, listItems is typically run by an analyst.

DS represents Π's whole state as a bit string. In practice, after adding n data *items* to Π with addItem, DS itself is internally organized as a collection of $L(n)$ internal data *values*. For example, a hash table consists of $L(n)$ cells, a tree consists of $L(n)$ nodes etc., but one can imagine other organizations. Representing Π's whole state, DS does not only contain the collection of internal data values, but also includes cryptographic keys and other data required for operations addItem and listItems.

This paper assumes that it is possible to estimate the maximum number of data items n that will ever have to be stored in Π in advance. Knowing an upper bound n for the number of data items is crucial: we will see later that our new coding fails if more than n items are added, and vastly overestimating n can hurt performance of listItems. While n (including a reasonable security margin) can be estimated in many scenarios based on past experience and heuristics, we conjecture that it might not be estimated in other scenarios — where our techniques would not be applicable.

Any data structure Π for storage must hold two properties. Informally, if you add $data$ with addItem, then listItems should *with high probability* output $data$ later. Similarly, if listItems outputs a sequence of data, then this data should have previously been added with addItem. We formally define soundness and completeness (in the absence of crashes or adversarial modifications) below.

Definition 1 (Soundness). For all $s, n \in \mathbb{N}, \eta \leq n$, and all sequences $(DS_0, sk) \leftarrow \mathsf{Init}(1^s, n)$, $(DS_1 \leftarrow (\mathsf{addItem}(DS_0, data_1), \ldots, DS_\eta \leftarrow \mathsf{addItem}(DS_{\eta-1}, data_\eta))$ we have

$$Pr[\mathsf{listItems}(DS_\eta, sk, n) = (data_1, \ldots, data_\eta)] = 1 - o(1).$$

Definition 2 (Completeness). For all $s, n \in \mathbb{N}, \eta \leq n$, and tuples $(DS_\eta, sk, data_1, \ldots, data_\eta)$ with $(data_1, \ldots, data_\eta) = \mathsf{listItems}(DS_\eta, sk)$, there exists $(DS_0, \ldots, DS_{\eta-1})$ such that

$$Pr[(DS_0, sk) \leftarrow \mathsf{Init}(1^s, n), DS_1 \leftarrow \mathsf{addItem}(DS_0, data_1), \ldots, DS_\eta \leftarrow \mathsf{addItem}(DS_{\eta-1}, data_{\eta-1})] = 1 - o(1).$$

In both definitions, probabilities are taken over random coins of Init, addItem, and listItems.

2.2 Crashes and Recovery

A *crash* is an event which modifies or deletes some number δ of Π's internal data values in DS. The exact amount δ can often be estimated in advance as it depends on system parameters such as the system's buffer cache size, physical disk cache size, and cache eviction rates. We now extend soundness and completeness to the case of crashes.

Definition 3 (δ-Recovery). Let DS_η be the result of the sequence of operations $((DS_0, sk) \leftarrow \mathsf{Init}(1^s, n), DS_1 \leftarrow \mathsf{addItem}(DS_0, data_1), \ldots, DS_\eta \leftarrow \mathsf{addItem}(DS_{\eta-1}, data_\eta))$, and let DS_η's internal state have $L(n)$ data values. Function $\Delta(DS_\eta, DS'_\eta)$ outputs δ, if state DS'_η is the result of modifying or deleting $\delta \leq L(n)$ internal data values in DS_η.

Storage data structure Π provides δ-recovery, *iff* for all DS_η and DS'_η with $\Delta(DS_\eta, DS'_\eta) \leq \delta$, the calls to $\mathsf{listItems}(DS_\eta, sk, n)$ in definitions 1 and 2 can be replaced by $\mathsf{listItems}(DS'_\eta, sk, n)$, but soundness and completeness still hold with probability $1 - o(1)$.

2.3 Adversary Model

We now discuss our target security requirements and define an adversary model. We assume that at some point a fully-malicious adversary $\mathcal{A}$ compromises the computer system hosting data structure Π. By compromise, we mean that $\mathcal{A}$ reads out all current memory (RAM, disk) contents and learns all possible cryptographic secrets. This includes the current bit representation DS of Π. Also, $\mathcal{A}$ controls the computer system from now on. That is, $\mathcal{A}$ might diverge from program execution and perform operations of their liking.

Overview. Informally, the security properties we want to guarantee are forward Confidentiality and forward Integrity (together abbreviated as *CI*). The combination of forward confidentiality and forward integrity is called forward security.

- The notion of forward **confidentiality** states that $\mathcal{A}$ cannot learn anything about data added to data structure Π *before* the time of compromise.
- Forward **integrity** describes the effect of $\mathcal{A}$'s possible modifications on state DS captured during compromise. Typically, $\mathcal{A}$ should not be able to tamper with data collected *before* the time of compromise without being detected.
- However, in our specific context of crashes, we augment forward integrity by a data **recovery** property. Without any adversarial behavior, a regular crash might change some of state DS. A crash is regular system behavior and not cause for any concern. Yet, even though a crash might change DS, we want to be able to recover log data collected before the crash. (In practice, this is often achieved by various techniques for data coding.)
  Interestingly, if $\mathcal{A}$'s tampering and their modifications to DS are "small", comparable to a regular system crash, the recovery property will also imply that $\mathcal{A}$'s modifications will not have an effect on the outcome of listItems regarding data items added *before* the time of compromise. So, listItems will correctly output all items added before the time of compromise. In case $\mathcal{A}$ modifes "large" amounts of DS, differently from a regular crash, we want to be able to at least detect these modifications (tamper evidence).

As we assume $\mathcal{A}$ to have privileged system access, they can do anything they want with DS, and the strongest integrity notion one can achieve (and we will achieve) is tamper evidence. Such an adversary, allowed to arbitrarily divert from a protocol, but wanting to avoid detection, is also called covert adversaries in the literature [2]. In our case, listItems should *either* output all data added before the time of compromise correctly *or* detect modifications and output $\perp$.

We now formalize our Confidentiality, Integrity, and Recovery intuition.

Security Definition. Consider experiment $\mathsf{Exp}^{\mathsf{CIR}}_{\mathcal{A},\Pi}(s,\delta)$ in Figure 1, where s denotes a security parameter and δ the number of internal data values $\mathcal{A}$ can modify. For sequence $seq = (data_1, \ldots, data_n)$ of data items, $\mathsf{PREFIX}(seq, \eta)$ outputs the first $\eta \leq n$ items $(data_1, \ldots, data_\eta)$.

In $\mathsf{Exp}^{\mathsf{CIR}}_{\mathcal{A},\Pi}(s,\delta)$, adversary $\mathcal{A}$ starts by specifying the maximum number of data items n that data structure Π should be able to store. $\mathcal{A}$ outputs two sequences of data items, one with η_0 items, and the other with η_1 items, $\eta_0, \eta_1 \leq n$. One of the two sequences is randomly chosen and added with addItem to an initially empty data structure Π. Then, $\mathcal{A}$ fully compromises the system hosting Π and learns the system's complete state and thus DS. $\mathcal{A}$ can tamper with DS in any way they want and output new state DS'. Finally, listItems lists DS' contents.

We require that $\mathcal{A}$ should not have any advantage in breaking confidentiality or integrity of data added before the time of compromise. Note that $\mathcal{A}$ has full access to DS at the time of compromise, but not to the initially generated auxiliary information sk. Only listItems will have access to sk. This setup reflects typical scenarios where a logging device adds

$\{n, (data_1^0, \ldots, data_{\eta_0}^0), (data_1^1, \ldots, data_{\eta_1}^1), st_{\mathcal{A}}\} \leftarrow \mathcal{A}(1^s)$;
// Let $\eta_0, \eta_1 < n$
$\{DS, sk\} \leftarrow \mathsf{Init}(1^s, n)$;
$b \xleftarrow{\$} \{0, 1\}$;
for $i = 1$ **to** η_b **do**
    $DS \leftarrow \mathsf{addItem}(DS, data_i^b)$;
$\{b', DS'\} \leftarrow \mathcal{A}(st_{\mathcal{A}}, DS)$;
confidentiality = False; integrity = False;
if $b \neq b'$ **then**
    confidentiality = True;
$result \leftarrow \mathsf{listItems}(DS', sk, n)$;
if $[\text{PREFIX}(result, \eta_b) = (data_1^b, \ldots, data_{\eta_b}^b)] \vee [result = \perp \wedge \Delta(DS, DS') > \delta]$ **then**
    integrity = True;
output {confidentiality, integrity};

Fig. 1. Experiment $\mathsf{Exp}^{\mathsf{CIR}}_{\mathcal{A},\Pi}(s, \delta)$

log entries until eventually another party, the analyst, receives all log entries and analyzes them. In Figure 1, $st_{\mathcal{A}}$ denotes $\mathcal{A}$'s internal state which $\mathcal{A}$ carries through the experiment.

Definition 4. Data structure $\Pi = (\mathsf{Init}, \mathsf{addItem}, \mathsf{listItems})$ provides $F(\cdot)$-CIR-security, *iff* for all PPT adversaries $\mathcal{A}$ and same-length data items, there exist function $F(\cdot)$ and negligible function ϵ such that the following two probabilities hold:

$$Pr[\mathsf{Exp}^{\mathsf{CIR}}_{\mathcal{A},\Pi}(s, \delta).\mathsf{confidentiality} = \mathsf{False}] = \frac{1}{2} + \epsilon(s)$$

$$Pr[\mathsf{Exp}^{\mathsf{CIR}}_{\mathcal{A},\Pi}(s, \delta).\mathsf{integrity} = \mathsf{False}] \leq F(\cdot).$$

Here, security parameter s is sufficiently large, and the probabilities are taken over the random coins of $\mathcal{A}$ and Π.

Discussion. Definition 4 captures forward confidentiality and forward integrity together with data recovery in a model fashion.

The first part of the security definition above addresses forward confidentiality. Even though the adversary gets access to the complete internal state with possible secrets and cryptographic keys, they cannot learn anything about data items added before the time of compromise. $\mathcal{A}$ cannot learn how many data items have already been added. As with standard definitions of confidentiality (IND-CPA), we require all items $data_i^0, data_j^1$ to have the same length.

The second part of the definition targets forward integrity and data recovery. If $\mathcal{A}$ has tampered with DS only within up to δ modifications to DS's internal data values, listItems should output all data from before the time of compromise. If $\mathcal{A}$ has modified more than δ, then listItems outputs $\perp$ indicating failure. In the real world, this corresponds to either recovering all data items inserted before the time of compromise or detecting an adversary who modifies more than δ internal values. As we can adjust δ to the expected modifications of a real crash, we will be able to recover from crashes, detect adversarial behavior beyond δ modifications, and cancel adversarial modifications less than δ.

Also observe that we limit $\mathcal{A}$ to add at most n data items in $\mathsf{Exp}^{\mathsf{CIR}}_{\mathcal{A},\Pi}(s, \delta)$. If $\mathcal{A}$ adds more than the n items initially configured, we allow listItems to fail (output $\perp$), but this does not correspond to any valid attack in the real-world.

The goal of definitions 1 and 2 is to make basic correctness properties explicit, while Definition 4 focuses on δ-Recovery and confidentiality in the face of adversarial modifications.

3 OUR APPROACH

High-Level Idea. Essentially, we offer a new data structure which allows to add new (log) data items and later list all of them. To add a new data item, we will encode it in a specific way before writing it to disk. The coding we use is customizable in that it allows listing ("recovering") all previously added data items as long as not more than a threshold parameter δ of the data structure is invalid, e.g., overwritten, corrupted, inconsistent or modified by an adversary. We then set parameter δ to a value which matches the amount of data possibly lost during a real crash of computer system running the logging service. Based on the computer's specifics such as file system cache or disks buffers, one can typically estimate and bound δ. As a result, in case of a real crash, we will be able to recover all previously logged data. Moreover, all adversarial modifications of up to δ data will be neutralized. If the adversary tampers with more than δ, we know that the corrupted data cannot be due to a real crash and thus provide tamper evidence.

Our approach has three key components: (1) as its underlying basis, a coding scheme with an online encoding property, i.e., encoding new symbols without knowledge of previous encodings, *constant* write operations per encoded symbol, and high probability of decodability for up to $\sqrt{n}$ erasures, (2) techniques to augment coding schemes to provide forward integrity and forward confidentiality, and (3) an abstract data structure, combining (1) and (2), that provides key secure logging operations of adding and listing logged items while provably guaranteeing properties of forward confidentiality, integrity, and recovery.

In the following, we gradually describe each of the components. After presenting the idea how to encode data and how to provide forward security, we sketch how this is integrated in data structure Π. Formal details with pseudo-code follow in Section 4.

3.1 Online Coding Scheme

Our coding scheme is formally defined through its generator matrix G. An uncoded binary vector u from Galois field $GF(2^n)$ of length n is coded into a codeword $v \in GF(2^m)$ of length m, such that $v = G \cdot u$. Matrix G is an $m \times n$ matrix where each column has exactly k *random* entries equal to 1 and the remaining ones equal 0. This coding scheme is very efficient to implement and extends to non-binary input symbols u specified one at a time. The idea for encoding is then to XOR each new data item to k distinct, randomly chosen cells in a table of m cells. To decode, one solves the system of linear equations, with the table of m cells defining the right-hand side and G the coefficients of the unknowns of the left-hand side.

One might note similarities to LDPC codes or other efficient erasure codes such as Fountain codes. However, as discussed in great detail in Section 8, such codes (Reed-Solomon, LDPC, LT, Raptor, Tornado etc.) cannot be used in this context. From a coding perspective, our code looks rather simple, yet it enjoys a computational complexity of encoding which is constant in the number of input symbols, and it supports online encoding. Moreover, it has strong decodability guarantees which we will prove.

3.2 Forward Security

One approach to store data with forward confidentiality and integrity builds on key chains, see, e.g., Bellare and Yee [4]. Two parties, the log device and an analyst, initially share a cryptographic key K_0. Both agree that a new key K_i is computed from previous key K_{i-1} by applying a PRF as $K_i = \mathsf{PRF}_{K_{i-1}}(\gamma)$ for some constant γ or by applying a cryptographic hash function $K_i = h(K_{i-1})$. To store data $item_i$, the log device computes K_i out of K_{i-1}, deletes K_{i-1}, and stores an authenticated encryption of $item_i$ with key K_i. To decrypt and check integrity, the analyst re-computes all K_i

starting from K_0. Forward confidentiality and integrity follow from the fact that an adversary can only learn a key K_i at the time of compromise and cannot rewind to previous keys.

Note that in the coding scheme above, to successfully decode, it is required for the analyst to know G, i.e., the k random locations where each data items was placed in the table. Therefore, the log device derives the k random locations using a PRG, seeded with the current key K_i to encode a symbol. The analyst can then replay random coins and deduce the k locations for each symbol. This is forward secure, because the adversary only learns the current key and cannot replay previously used random locations. Along the same lines, forward security automatically implies that the coding scheme must allow online encoding: information about random locations chosen previously is not available anymore after an encoding.

3.3 Abstract Data Structure

We now combine coding and forward security techniques and apply them into a new data structure Π. First, we use authenticated encryption with the current key to encrypt the current data item. A standard trick is then to evolve the key which automatically yields forward confidentiality and forward integrity as with related work [37]. Then, our main idea and contribution is a new way to encode the resulting ciphertext. We XOR the ciphertext to k distinct, pseudo-randomly chosen cells in a table of m cells. After adding n data items we decode (table cells represent right-hand side of system of linear equations, cell indices represent left-hand side). For certain choices of $m > n$ and k, we will prove that the resulting system of equations has rank n even if we remove up to δ equations from it. Thus, we use Gaussian elimination to recover all n data items previously added.

We now give a more technical overview over our data structure Π, with full technical details following in Section 4. Let $k \in \mathbb{N}$ be a small system parameter.

Authenticated Encryption AE$. Let Algorithms $(\mathsf{Enc}_K, \mathsf{Dec}_K)$ realize authenticated encryption as follows: the encryption part is IND$-CPA [36] secure, and integrity tags are generated by a pseudo-random function family. We call the resulting authenticated encryption, where the whole bit string output by Enc_K is indistinguishable from a random string, AE$ encryption. A standard real-world example for AE$ encryption is encrypt-then-MAC with AES-CTR and HMAC. Let function PRF_K specify a pseudo-random function family indexed by key K. In practice, HMAC might serve as such a PRF. Finally, $\mathsf{PRG}(s)$ denotes a pseudo-random generator with seed s, e.g, an AES-CTR based construction.

Simplification: Single Key. To keep our exposition simple, we omit an important security detail and come back to it later. For now, our exposition uses the same key K_i as a key for different cryptographic primitives, e.g., Enc, PRF, and PRG. While this eases understanding of our main techniques, it also leads to unclear security. Instead, one should employ separate keys for each primitive, and we show how to do that later in Section 4.4. Note that, below, we also use a PRF with inputs of different lengths. This is possible with, e.g., HMAC, but other PRFs might require appropriate padding.

3.3.1 Initialization. First, assume that the maximum number of data items ever to be added into our data structure can be estimated and upper bound by n. Also, assume that all data items have a maximum (padded) length of ℓ bits. During initialization of Π, Init creates an empty table $\mathcal{T}_0$ with $m > n$ cells $\mathcal{T}_0[i]$. Init also generates the start of a key chain $K_0 \xleftarrow{\$} \{0,1\}^s$. The initial state (bit representation) of Π is $DS_0 = (\mathcal{T}_0, K_0)$, and Init outputs (DS_0, K_0). In our case with a table, the internal data values of our data structure are the table's cells; the number $L(n)$ of internal data values is $L(n) = m$.

3.3.2 Adding data to Π. At the beginning of the i^{th} invocation of addItem, Π's state is $DS_i = (\mathcal{T}_i, K_i)$. When adding length ℓ bit item $data_i$, the following modifications to $\mathcal{T}_i$ turn it into $\mathcal{T}_{i+1}$. First, we encrypt $data_i$ to $c_i \leftarrow \mathsf{Enc}_{K_i}(data_i)$.

$$AM = \left[\begin{array}{cccc|c} c_1 & c_2 & \cdots & c_n & \mathcal{T}_n \\ \hline 0 & 0 & \cdots & & \mathcal{T}_n[1] \\ \vdots & 1 & \vdots & & \vdots \\ 0 & 0 & & & \\ 1 & \vdots & & & \\ 0 & 0 & & & \\ \vdots & 1 & & & \\ 0 & 0 & & & \\ 1 & \vdots & & & \\ 0 & & & & \vdots \\ \vdots & & & & \mathcal{T}_n[m] \end{array}\right]$$

Fig. 2. Augmented matrix $AM = [M_n|\mathcal{T}_n]$

We randomly choose k distinct cells in $\mathcal{T}_i$. Table $\mathcal{T}_i$ becomes $\mathcal{T}_{i+1}$ by XORing c_i to each of these k cells. Finally, we compute K_{i+1} out of K_i and discard K_i.

3.3.3 Listing all data items. Knowledge of K_0 permits deriving all K_i and cell indices, and we exploit this for listItems. That is, knowing K_0 permits listItems to replay all random coins which PRG produced during addition of data items, thus listItems knows the indices in which cells each data item $data_i$ has been XORed to. Recall that XORing equals addition in finite fields of characteristic 2. So, listItems can set up a system of linear equations $M_n \cdot \vec{x} = \mathcal{T}_n$. Each cell of the table represents one component of the vector. We show the augmented matrix $AM = [M_n|\mathcal{T}_n]$ in Figure 2. Each column vector of M_n specifies where ciphertext c_i has been XORed to $\mathcal{T}_n$. Note that each column vector has constant weight k.

Algorithm listItems solves the system of linear equations AM to receive candidates for c_i. Then, it verifies the integrity tag for each candidate c_i and outputs c_i if verification succeeds. The key observation is that as long as M_n has rank n, Gaussian elimination will always succeed and output sequence $(c_1, \ldots, c_n)$. We will prove in Section 5 that M_n has rank n with high probability, even if δ equations are removed from M_n and $\mathcal{T}_n$.

Note that there are several optimizations possible which we discuss later in Section 5. We will also discuss the exact choice of parameters c and k there.

3.3.4 Caveats. Before concluding our high level overview and turning to technical details, we emphasize the need for several additional techniques to make this approach secure.

- First, there must be a way for listItems to verify whether the i^{th} cell of table $\mathcal{T}_j$ is broken, i.e., whether a crash has overwritten parts of the cell's content. Otherwise, a broken bit sequence in $\mathcal{T}_j[i]$ will lead to an inconsistent system of linear equations. To mitigate, we extend each cell in the table by an additional integrity tag T_i. Specifically, each cell comprises as its first part XOR_i the XOR of AE$ encryptions and as a second part an integrity tag T_i computed over the first part. During insertion of $data_j$, after addItem has XORed c_j to first part XOR_i of cell $\mathcal{T}_j[i]$, we write $\text{PRF}_{K_j}(\text{XOR}_i)$ into the second part T_i of $\mathcal{T}_j[i]$. Later, algorithm listItems will remove all equations from AM where the PRF part does not match the XOR_i part. Adversary $\mathcal{A}$ can arbitrarily modify contents of cell $\mathcal{T}_j[i]$ and then compute a

Algorithm 1: $\mathsf{Init}(1^s, n)$

// Let $c > 1, k > 2, \gamma \xleftarrow{\$} \{0,1\}^s$
1 $m = c \cdot n$;
2 $\mathcal{T}_0 = \mathsf{EmptyTable}(m, \ell + 4 \cdot s)$;
3 $K_0 \xleftarrow{\$} \{0,1\}^s$;
4 $pad = \mathsf{PRG}(K_0)$;
5 $\mathcal{T}_1 = \mathcal{T}_0 \oplus pad$;
6 $K_1 = \mathsf{PRF}_{K_0}(\gamma)$;
7 $DS_1 = (\mathcal{T}_1, K_1)$;
8 **output** (DS_1, K_0);

Algorithm 2: $\mathsf{addItem}(data_i, DS_i)$

// Parse DS_i as $(\mathcal{T}_i, K_i)$
1 $c_i \leftarrow \mathsf{Enc}_{K_i}(data_i)$;
// Generate Distinct Random Numbers (DRN)
2 $\{l_1, \ldots, l_k\} = \mathsf{DRN}(\mathsf{PRG}(K_i))$;
3 $\mathcal{T}_{i+1} = \mathcal{T}_i$;
4 **foreach** l_j **do**
  // Parse $\mathcal{T}_{i+1}[l_j]$ as $(\mathsf{XOR}_{l_j}, T_{l_j}, ID_{l_j})$, let $\gamma' \xleftarrow{\$} \{0,1\}^s$ be constant
5 $\mathsf{XOR}_{l_j} = \mathsf{XOR}_{l_j} \oplus c_i$;
6 $T_{l_j} = \mathsf{PRF}_{K_i}(\mathsf{XOR}_{l_j})$;
7 $ID_{l_j} = \mathsf{PRF}_{K_i}(\gamma', j)$;
8 $K_{i+1} = \mathsf{PRF}_{K_i}(\gamma)$;
9 **output** $DS_{i+1} = (\mathcal{T}_{i+1}, K_{i+1})$

T_i using their current K_j. However, if $\mathcal{A}$ introduces inconsistencies in AM, listItems detects that such inconsistencies cannot be a result of a crash.

- For listItems to understand which key K_j was used to generate tag T_i, we also store $ID_i = \mathsf{PRF}_{K_j}(\gamma')$ in $\mathcal{T}_j[i]$, where γ' is another constant different from γ. So, listItems can once generate all n possible IDs, store them in a separate hash table, and then access them in expected time $O(1)$. In conclusion, each cell $\mathcal{T}_j[i]$ comprises XOR_i, T_i, and ID_i.
- Table $\mathcal{T}_0$ cannot be empty (filled with zeros) after initialization. Otherwise, $\mathcal{A}$ would be able to distinguish between empty and non-empty cells in $\mathcal{T}_n$ with some probability. $\mathcal{A}$ would then be to focus on deleting non-empty cells from the table, violating forward integrity with high probability. Consequently, we initialize $\mathcal{T}_0$ by filling it pseudo-randomly. We start with a random K_0 and fill $\mathcal{T}_0$ with the output of $\mathsf{PRG}(K_0)$, which results in $\mathcal{T}_1$. Later, listItems removes initial pseudo-randomness by re-computing it and XORing to $\mathcal{T}_n$.

4 DETAILED DESCRIPTION

4.1 Init

For an estimated upper bound of n data items, Algorithm 1 (Init) generates table $\mathcal{T}$ with m cells, where $m = c \cdot n, c > 1$. Init also generates a key K_0. The total size of each cell in $\mathcal{T}$ is $\ell + 4 \cdot s$ bit. The first $\ell + 2 \cdot s$ bit are reserved for AE$ encryption of a size ℓ bit data item and include the encryption's random coins and integrity tag (e.g., AES-CTR and

HMAC). We call the first $\ell + 2 \cdot s$ bit of the cell the XOR part, as this is what will be XORed during addItem. Each cell also includes s bit for an additional integrity tag T and another s bit for a key ID. In conclusion, the i^{th} cell $\mathcal{T}[i]$ comprises $(\text{XOR}_i, T_i, ID_i)$.

Init fills cells with the output of $\text{PRG}(K_0)$, see Line 5 of Algorithm 1. Finally, Init outputs Π's state DS which is the table, the next key K_1, and the number of data items in Π, i.e., 0.

4.2 addItem

To add a new data item $data_i$ to data structure Π, which already holds $i-1$ data items, Algorithm 2 (addItem) first uses AE$ encryption and encrypts $data_i$ using current key K_i to ciphertext c_i. Then, addItem pseudo-randomly chooses k distinct indices l_j, for $1 \leq l_j \leq m$, in table $\mathcal{T}$. To generate required (pseudo-)randomness, addItem uses pseudo-random generator PRG with K_i as seed. For each cell $\mathcal{T}[l_j]$ indexed by l_j, addItem XORs c_i to $\mathcal{T}[l_j]$'s XOR part. It then computes PRF_{K_i} over this XOR part and stores the result as integrity tag T_{l_j} in $\mathcal{T}[l_j]$. Finally, to later help listItems to find correct key K_i for decryption, addItem also stores key ID $ID_{l_j} = \text{PRF}_{K_i}(\gamma', j)$ in $\mathcal{T}[l_j]$. Here, "," is an unambiguous pairing of inputs such as concatenation of fixed-length j and constant γ'. Operation addItem outputs the updated table and a new key K_{i+1}.

Note that key IDs ID_{l_j} are part of table $\mathcal{T}$ and thus obtained by adversary $\mathcal{A}$ at the time of compromise. This is not an issue as IDs are not secret. They are the output of a PRF, so for an adversary without knowledge of (previous) key K_i, a ID is indistinguishable from a random bit string of the same length.

As with IND-CPA encryption, also AE$ encryption requires all data items to have the same length ℓ. In practice, data items might therefore require padding.

Replacing K_i by K_{i+1} enables forward confidentiality and forward integrity. If $\mathcal{A}$ compromises after i data items have been added, they will learn K_{i+1}. Thus, $\mathcal{A}$ cannot modify anything that was encrypted with a key K_j for $j \leq i$. Similarly, indices previously generated by PRG are indistinguishable from random for $\mathcal{A}$.

4.3 listItems

Algorithm 3 (listItems) recovers all data items from DS. It receives initial key K_0 as a parameter and starts by re-computing all n possible keys K_i. To be able to link key K_i with its k corresponding IDs, listItems uses a simple (key,value) store KeyStore. For all of the k possible locations l_j where K_i could have been used to compute T_{l_j}, it stores tuple (K_i, i, l_j) under its key ID in KeyStore.

The main idea now is to compute $m \times n$ matrix M, the matrix of coefficients over $GF(2)$ representing the left-hand side of the system of linear equations. M is initially all zero. As a first step, listItems computes the number of log entries which have been added to Π, and thus M's expected *rank*, by checking which key IDs and which keys have been used (Line 11).

Then, using keys K_i, listItems replays all indices for item $data_i$. For each position l_j in $\mathcal{T}$, listItems puts a 1 in column i, row l_j of M. However, it does this only, if the corresponding integrity tag T_{l_j} matches XOR_{l_j}, i.e., if this table cell was not modified by the adversary or during a crash. To check T_{l_j}, listItems fetches the corresponding key from KeyStore using ID_{l_j}. Note that listItems also verifies whether content in $\mathcal{T}[l_j]$ is supposed to be at position l_j (Line 18). If one of the checks fails, coefficient $M[l_j, i]$ remains 0, and XOR_i is set to $0^{\ell+2\cdot s}$. We essentially remove this equation from the system of equations. To remove the initial randomness from all XOR_i, listItems re-computes the random bit string pad and XORs it to all remaining (non-zero) cells in $\mathcal{T}$. We represent resulting $\mathcal{T}$ as a vector $\vec{v}$ of dimension m.

Algorithm 3: listItems(DS, K_0, n)

```
  // Parse DS as (T, K_η) and T[i] as (XOR_i, T_i, ID_i), 1 ≤ i ≤ m
  // Let KeyStore be a (key,value)-pair data structure.
  // Generate all possible keys and IDs, store in KeyStore.
1 for i = 1 to n do
2 |   K_i = PRF_{K_{i-1}}(γ);
  |   // Re-Generate k Distinct Random Numbers (DRN)
3 |   {l_1, ..., l_k} = DRN(PRG(K_i));
4 |   for j = 1 to k do
5 |   |   ID = PRF_{K_i}(γ', j);
6 |   |   KeyStore.put(ID, (K_i, i, l_j));
7 M=m × n zero matrix over GF(2);
  // Predict M's rank rank
8 rank = 0;
9 for i = 1 to m do
10|   (K_j, j, ℓ) = KeyStore.get(ID_i);
11|   rank = max(rank, j);
12 if rank = 0 then output ⊥;
13 else
14|   for i = 1 to rank do
15|   |   {l_1, ..., l_k} = DRN(PRG(K_i));
16|   |   for j = 1 to k do
17|   |   |   (K_t, t, l) = KeyStore.get(ID_{l_j});
18|   |   |   if l = l_j and PRF_{K_t}(XOR_{l_j}) = T_{l_j} then M[l_j, i] = 1; else XOR_{l_j} = 0^{ℓ+2·s};
  |   // Re-generate m(ℓ + 4s) bit pseudo-random pad
19|   pad = PRG(K_0); // Let pad[i] denote the ℓ + 2·s bit string covering XOR_i in T[i].
20|   for i = 1 to m do
21|   |   if XOR_i ≠ 0^{ℓ+2·s} then XOR_i = XOR_i ⊕ pad[i]
  |   // Let vector v = (XOR_1, ..., XOR_m) ∈ GF(2^{(ℓ+2·s)^m})
  |   // Let vector c be a solution vector over (GF(2^{ℓ+2·s}))^n
22|   Solve M · c = v for c;
23|   if equations are inconsistent then output ⊥;
24|   else //Let c_1, ..., c_rank be the solutions
25|   |   for i = 1 to rank do
26|   |   |   data_i = Dec_{K_i}(c[i]);
27|   |   |   if data_i ≠ ⊥ then output data_i;
```

Finally, listItems solves the resulting system of linear equations $M \cdot \vec{c} = \vec{v}$, e.g., by applying Gaussian elimination. This results in solution $\vec{c}$ which listItems decrypts componentwise to get $data_i$. If Gaussian elimination finds an inconsistent system of equations, listItems outputs $\perp$. Otherwise, it outputs $(data_1, \ldots, data_{rank})$.

4.4 Additional Security Details

To keep our exposition clear, we have omitted two important details.

Separate Keys. In our description, we have used the same cryptographic key for different cryptographic primitives to simplify notation. However, one must use different keys for each cryptographic primitive, i.e., K_i^{X} for primitive $\mathsf{X} \in \{\mathsf{Enc}, \mathsf{PRF}, \mathsf{PRG}\}$, to allow black-box security arguments for each primitive. Also, instead of just evolving K_i to K_{i+1} each of these keys must be evolved by computing $K_{i+1}^{\mathsf{X}} = \mathsf{PRF}_{K_i^{\mathsf{X}}}(\gamma)$. Another alternative would be to each time derive key K_i^{X} for primitive X from key K_i using a derivation function such as $K_i^{\mathsf{X}} = \mathsf{PRF}_{K_i}(\gamma_{\mathsf{X}})$ and different constants γ_{X}. Only K_i would be evolved to K_{i+1}.

Minimum Rank. The system of equations which we are building needs to have a minimum rank. Note that listItems of Algorithm 3 would output $\perp$ for a freshly initialized data structure. That is, $(DS, K) \leftarrow \mathsf{Init}(1^s, n), \mathsf{listItems}(DS, K, n)$ returns $\perp$. Moreover, during compromise, adversary $\mathcal{A}$ can change state DS to a garbage state DS' by overwriting DS with random bit strings. A completely random DS' violates the requirement of up to δ modifications or deletions. As listItems will not find any valid ID, it would set $rank$ to 0, M would remain a $m \times n$ zero matrix, and listItems would output an empty set, but not $\perp$.

However, we remedy this issue by simply adding a single dummy data item during initialization. We create a table for $n + 1$ data items and then call addItem once to add a dummy data item. If listItems detects a rank of 0, it outputs $\perp$.

5 ANALYSIS

We will now prove that data structure Π provides $o(1)$-CIR-security. So, for function F from Definition 4, we have $F(n) = o(1)$.

Roadmap. The idea behind proving this is to show that Π is sound, complete, and offers δ-Recovery up to the security of underlying cryptographic primitives.

As our key chain techniques for forward integrity and confidentiality are rather standard and rely on standard cryptographic constructions, we simply dismiss adversaries $\mathcal{A}$ violating confidentiality ($\mathsf{Exp}^{\mathsf{CIR}}_{\mathcal{A},\Pi}(s, \delta)$.confidentiality is True) or integrity ($\mathsf{Exp}^{\mathsf{CIR}}_{\mathcal{A},\Pi}(s, \delta)$.integrity is True) by breaking cryptographic primitives. Let s be the security parameter for cryptographic primitives PRG, PRF, and Enc used in their respective security definitions. We summarize that the probability of $\mathcal{A}$ breaking forward integrity or confidentiality by attacks on cryptographic primitives is negligible in s. Also note that $\epsilon(s) \in o(1)$.

Thus, the **focus of our analysis** is to formally prove crash recovery properties. That is, even in case of a crash or an adversary tampering with up to δ data items, δ-Recovery of Definition 3 holds, so the original data can be recovered with high probability. As a warm-up, we first prove that, without a crash or adversarial behavior, Π is both sound and complete (§5.1). We then turn to the main contribution of our analysis, the proof of δ-Recovery (§5.2).

5.1 Soundness and Completeness

Lemma 1. For any $k \geq 2$, there is a $c \in O(1)$ such that data structure $\Pi = (\mathsf{Init}, \mathsf{addItem}, \mathsf{listItems}, DS)$ is sound and complete with probability $1 - o(1)$.

PROOF. The key insight is that our way of constructing M corresponds to adding random binary length m vectors with exactly k "1"s to an initially empty set S. There exist several fundamental results quantifying a threshold size n of S, $n = |S|$, until which vectors remain linearly independent and starting from which vectors in S become linearly dependent with high probability [9, 10, 13, 14].

For $k = 2$, if $m = c \cdot n$ and $c > 2$, the n vectors in S remain linearly independent almost surely with increasing m and n, i.e., with high probability $1 - o(1)$ [14].

For each $k \geq 3$, it has been shown that there exists a c such that, as long as $n < m/c$, the n vectors in S remain linearly independent with probability $1 - o(1)$, see Theorem 2b in Calkin [10]. Moreover, c can be approximated by $c^{-1} \approx 1 - \frac{e^{-k}}{\ln 2} - \frac{e^{-2 \cdot k}}{2 \cdot \ln 2} \cdot (k^2 - 2 \cdot k + \frac{2 \cdot k}{\ln 2} - 1)$. With increasing k, values for c go (exponentially fast) towards 1.

A linear independent set S directly translates to our $m \times n$ matrix M having rank n. If M's rank is n, listItems solves the related system of linear equations with probability 1. If listItems can solve the system of linear equations, soundness (Definition 1) and completeness (Definition 2) follow immediately. □

Building on these results, we set $k \geq 3$ for the remainder of the paper. Varying k allows for tailoring storage requirements and computational overhead. For example, Dietzfelbinger and Pagh [13] present that for $k = 5$ we must set $c > 1.011$. With such a configuration, set S remains linear independent (and thus M has rank n) with probability $1 - O(\frac{1}{n})$. Moreover, it gives reasonable real-world performance: 5 calls to a PRG and accesses to the table per addItem and a small storage overhead of only 1%.

5.2 δ-Recovery and Choice of c and k

More challening is to show that the rank of M remains equal to n, as long as $c = \frac{m}{n}$ is greater than a threshold c_0, and the number δ of rows removed from M is bound by $\sqrt{n}$. Following Definition 3, full rank despite removal of $\sqrt{n}$ rows automatically implies that Π provides $\sqrt{n}$-recovery. This is our main result and stated in Corollary 1.

While it is possible to show the existence of a phase transition phenomena (i.e., a necessary and sufficient condition for being able to decode all data items), in this paper we focus on a bound of $c = \frac{m}{n}$ that guarantees recovery (decodablity). The proof technique follows Calkin's proof for analyzing the rank of a binary matrix with columns of constant weight [10]. We extend the proof technique to the case of columns with hypergeometrically distributed weight. The hypergeometric distribution is the result of deleting arbitrary δ rows from the original matrix.

The proof outline is as follows. First, we consider Markov chain MH defined by the random walk on the hypercube 2^m using vectors of hypergeometric weight. In Theorem 2, we show that the expected rank of matrix M directly derives from the eigenvalues of MH's transition matrix (called H). We establish bounds for the eigenvalues of H, for the considered values of k, δ, and c, that lead to the asymptotic rank guarantee. In this section, k refers to the constant weight of the generator matrix column vectors, i.e., the number of distinct cells each data item gets XORed to. Later, we will set $k = 5$.

Let $S_{m,k,\delta}$ denote the set of vectors over $GF(2)$ of length m and Hamming weight $k-\kappa$, κ distributed hypergeometrically. We have

$$Pr[\text{weight}(u \in S_{m,k,\delta}) = k - \kappa] = \frac{\binom{k}{k-\kappa}\binom{m-k}{\delta-k+\kappa}}{\binom{m}{\delta}}.$$

M can be viewed as a matrix of n columns $u_1, u_2, \ldots, u_n$, chosen randomly from $S_{m,k,\delta}$. Let r be the rank of M, and the difference between n and rank r is $d = n - r$. We write $E(\cdot)$ for the expectation of a random variable. Our main result, Corollary 1, directly derives from the following Theorem 1.

Theorem 1. If $\delta < \sqrt{n}$, then there exists $c_0 > 1$ such that

$$\text{if } c > c_0 \text{ and } m > c \cdot n \quad \Longrightarrow \quad E(2^d) \to 1 \text{ as } m \to \infty.$$

PROOF. This is the main theorem that we will prove in several steps below. □

Corollary 1 (δ-Recovery). If $\delta < \sqrt{n}$, there exists $c_0 > 1$ such that if $c > c_0$ and $m > c \cdot n \implies Pr[\text{rank}(M) = n] = 1 - o(1)$.

PROOF. This derives immediately from Theorem 1, since $E(2^d) \to 1$ (when $n \to \infty$) implies that $d \to 0$, and therefore the rank $r \to n$. Finally, if matrix M has rank n, then we recover all n data items with Gaussian elimination, which results in δ-recovery. □

We now turn to Theorem 1. To prove this theorem, we first define a random walk on the 2^m hypercube using steps u_i of hypergeometrically distributed weight. Let the random variable describing the position on the hypercube be x_i, and $x_0 = 0$ and $x_i = x_{i-1} + u_i$.

We introduce the Markov chain MH with state defined by the weight of x_i.

Lemma 2. The transition matrix H of MH has the following two properties:

1. $H = \sum_{\kappa=0}^{k} Pr[\text{weight}(u_i) = \kappa] \cdot A^{(\kappa)}$, where $A^{(\kappa)}$ is the transition matrix for the random walk Markov chain given by u_i of constant weight κ.
2. $H_{(p,q)} = \sum_{\kappa=0}^{k} \frac{\binom{k}{k-\kappa}\binom{m-k}{\delta-k+\kappa}}{\binom{m}{\delta}} \frac{\binom{q}{\frac{\kappa-p+q}{2}}\binom{m-q}{\frac{\kappa+p-q}{2}}}{\binom{m}{\kappa}}$, for $0 \le p, q \le k$ and where the binomial coefficients are interpreted to be 0, if $\kappa + p + q$ is odd.

PROOF. $H_{(p,q)}$ denotes the probability of transitioning from state p to state q. Note that here, when we are in state p, we add a random vector u_i of hypergeometrically distributed weight k. Therefore, $H_{(p,q)}$ is basically the sum of the probability of u_i having a given weight κ *times* the probability that we transition from state p to state q with fixed weight κ (which is by definition $A^{(\kappa)}_{(p,q)}$ and is equal to $\frac{\binom{q}{\frac{\kappa-p+q}{2}}\binom{m-q}{\frac{\kappa+p-q}{2}}}{\binom{m}{\kappa}}$).

$$\begin{aligned} H_{(p,q)} &= \sum_{\kappa=0}^{k} Pr[\text{weight of } u_i \text{ is } \kappa] \cdot A^{(\kappa)}_{(p,q)} \\ &= \sum_{\kappa=0}^{k} \frac{\binom{k}{k-\kappa}\binom{m-k}{\delta-k+\kappa}}{\binom{m}{\delta}} A^{(\kappa)}_{(p,q)} \\ &= \sum_{\kappa=0}^{k} \frac{\binom{k}{k-\kappa}\binom{m-k}{\delta-k+\kappa}}{\binom{m}{\delta}} \frac{\binom{q}{\frac{\kappa-p+q}{2}}\binom{m-q}{\frac{\kappa+p-q}{2}}}{\binom{m}{\kappa}} \end{aligned} \tag{1}$$

Equation 1 also implies $H = \sum_{\kappa=0}^{k} Pr[\text{weight}(u_i) = \kappa] \cdot A^{(\kappa)}$. □

Lemma 3. H's eigenvalues λ_i^H are a linear combination of the eigenvalues $\lambda_{\kappa,i}$ of the constant weight (κ) transition matrices. Formally,

$$\lambda_i^H = \sum_{\kappa=0}^{k} \frac{\binom{k}{k-\kappa}\binom{n-k}{\delta-k+\kappa}}{\binom{n}{\delta}} \lambda_{\kappa,i} \tag{2}$$

$$\lambda_{\kappa,i} = \sum_{t=0}^{\kappa} (-1)^t \frac{\binom{i}{t}\binom{m-i}{\kappa-t}}{\binom{m}{\kappa}} \tag{3}$$

$$e_i[j] = \sum_{t=0}^{j} (-1)^t \binom{i}{t}\binom{m-i}{j-t} \tag{4}$$

PROOF. From [10] (Lemma 2.2), $A^{(\kappa)} = \frac{1}{2^m} U \Delta^{(\kappa)} U$, where U is defined by columns (eigenvectors) e_i, and Δ is the diagonal eigenvalues matrix defined by λ_i^H. We also note the following property:

$$U^2 = 2^m I. \tag{5}$$

$$\begin{aligned}\text{Then, } H &= \sum_{\kappa=0}^{k} \frac{\binom{k}{k-\kappa}\binom{m-k}{\delta-k+\kappa}}{\binom{m}{\delta}} A^{(\kappa)} \\ &= \sum_{\kappa=0}^{k} \frac{\binom{k}{k-\kappa}\binom{m-k}{\delta-k+\kappa}}{\binom{m}{\delta}} \frac{1}{2^m} U\Delta^{(\kappa)}U \\ &= \frac{1}{2^m} U(\sum_{\kappa=0}^{k} \frac{\binom{k}{k-\kappa}\binom{m-k}{\delta-k+\kappa}}{\binom{m}{\delta}} \cdot \Delta^{(\kappa)})U = \frac{1}{2^m} U\Lambda U\end{aligned}$$

where $\Lambda = \sum_{\kappa=0}^{k} \frac{\binom{k}{k-\kappa}\binom{m-k}{\delta-k+\kappa}}{\binom{m}{\delta}} \cdot \Delta^{(\kappa)}$. As $\Delta^{(\kappa)}$ is diagonal, Λ is diagonal, and $\frac{1}{2^m} U\Lambda U$ is the eigen-decomposition of H with eigenvectors the columns of U and eigenvalues $\sum_{\kappa=0}^{k} \frac{\binom{k}{k-\kappa}\binom{n-k}{\delta-k+\kappa}}{\binom{n}{\delta}} \lambda_{\kappa,i}$. □

Remark. Eigenvectors of H do not depend on κ and form matrix U. It therefore does not matter if we take a step of size z and then z' or z' and then z. The probability that $u_1, u_2, \ldots, u_t$ sum to 0 is the 00^{th} coefficient of H^t. So, considering all possible combinations of t columns of u_i, and given the fact that we are operating in $GF(2)$, the expected number of combinations of u_i that add-up to 0, we derive

$$E(2^d) = \sum_{t=0}^{m} \binom{m}{t} (H^t)_{00}.$$

Theorem 2.

$$E(2^d) = \sum_{t=0}^{m} \frac{1}{2^m} \binom{m}{i} (1 + \lambda_i^H)^n$$

PROOF. We already know that $H = \frac{1}{2^m} U\Lambda U$. We can therefore derive H^t:

$$H^t = (\frac{1}{2^m} U\Lambda U)^t = \frac{1}{2^m} U\Lambda^t U \tag{6}$$

Equation 6 derives from equation 5, which states that $U \cdot U = 2^n I$.

Given that U is defined by the eigenvectors e_i, the 00^{th} coefficient of H^t can be calculated as:

$$(H^t)_{00} \quad = \quad \sum_{i=0}^{m} \frac{1}{2^m} (\lambda_i^H)^t \binom{m}{i}$$

Since u_i are chosen randomly and independently, the expected number of subsequences that add up to 0 is:

$$\begin{aligned}E(2^d) &= \sum_{t=0}^{n} \binom{n}{t} (H^t)_{00} = \sum_{t=0}^{n} \binom{n}{t} \sum_{i=0}^{m} \frac{1}{2^m} (\lambda_i^H)^t \binom{m}{i} \\ &= \sum_{i=0}^{m} \frac{1}{2^m} \binom{m}{i} (1 + \lambda_i^H)^n\end{aligned}$$

□

Lemma 4. H's eigenvalues satisfy the following:

(1) $\forall 0 \leq i \leq n$: $|\lambda_i^H| < 1$

(2) $\forall t < \frac{1}{2}$, if $i = t \cdot m$, then

$$\lambda_i^H < (1 - \frac{2i}{m})^3 - \frac{4\binom{k}{2}}{m}(1 - \frac{2i}{m})\frac{i}{m}(1 - \frac{i}{m}) + O(\frac{k^3}{c^2 m^2}) + O((\frac{\delta}{m})^3)$$

(3) For i sub-linearly close to $\frac{m}{2}$, i.e., $\frac{m}{2} - i = \frac{m^\theta}{2}$ (for some $\theta < 1$), we have

$$\lambda_i^H < (\frac{1}{m^{1-\theta}})^3 - \frac{4\binom{k}{2}}{m}(\frac{1}{m^{1-\theta}}) + O(\frac{k^3}{m^2}) + O((\frac{\delta}{m})^3) \quad (7)$$

PROOF. The first inequality derives from the formulae of λ_i^H, and $\lambda_{k,i}$ as shown in Lemma 3, equations 2 and 3. The second inequality derives from the bound of $\lambda_{\kappa,i}$ as derived by Lemma 3.1(c) in [10], combined with Equation 2 setting k to 5. The last inequality derives from replacing $\frac{m}{2} - i$ by $\frac{m^\theta}{2}$. □

Corollary 2. For $\delta < m^\gamma$ where $\gamma < \frac{2}{3}$, if $\theta < \frac{2}{3}$ and $\frac{m}{2} - i = \frac{m^\theta}{2}$, then $\lambda_i^H m \to 0$ as $m \to +\infty$.

PROOF. This derives from Equation 7. □

This is in particular true for $\gamma = \frac{1}{2}$, and we therefore set $\delta = \sqrt{n}$.

Lemma 5. There exists $c_0 > 1$ such that if $c > c_0$ and $m > c \cdot n$ then $\exists \epsilon$ such that:

(1) $A = \sum_{i=0}^{\epsilon m} \frac{1}{2^m}\binom{m}{i}(1 + \lambda_i^H)^n + \sum_{i=(1-\epsilon)m}^{m} \frac{1}{2^m}\binom{m}{i}(1 + \lambda_i^H)^n \to 0$ as $m \to +\infty$

(2) $B = \sum_{i=\frac{m}{2}-m^{\frac{4}{7}}}^{i=\frac{m}{2}+m^{\frac{4}{7}}} \frac{1}{2^m}\binom{m}{i}(1 + \lambda_i^H)^n \to 1$ as $m \to +\infty$

(3) $C = \sum_{i=\frac{m}{2}(1-\epsilon)}^{\frac{m}{2}-m^{\frac{4}{7}}} \frac{1}{2^m}\binom{m}{i}(1 + \lambda_i^H)^n + \sum_{i=\frac{m}{2}+m^{\frac{4}{7}}}^{\frac{m}{2}(1+\epsilon)} \frac{1}{2^m}\binom{m}{i}(1 + \lambda_i^H)^n \to 0$ as $m \to +\infty$

(4) $D = \sum_{i=\epsilon m}^{\frac{m}{2}(1-\epsilon)} \frac{1}{2^m}\binom{m}{i}(1 + \lambda_i^H)^n + \sum_{i=\frac{m}{2}(1+\epsilon)}^{(1-\epsilon)m} \frac{1}{2^m}\binom{m}{i}(1 + \lambda_i^H)^n \to 0$ as $m \to +\infty$

Moreover, $c_0 \approx 1.1243$.

PROOF OF LEMMA 5. Since eigenvalues λ^H of H satisfy the same asymptotic bounds as λ_i in the case of [10], the results derive similarly. Note that (1), (2), and (3) are always true independently of c_0. (4) is also, but requires c_0 to be a solution to a function f defined in the next steps.

Given that $|\lambda_i^H| < 1$,

$$A = \sum_{i=0}^{\epsilon m} \frac{1}{2^m}\binom{m}{i}(1 + \lambda_i^H)^n + \sum_{i=(1-\epsilon)m}^{m} \frac{1}{2^m}\binom{m}{i}(1 + \lambda_i^H)^n < \sum_{i=0}^{\epsilon m} \frac{1}{2^{m-n-1}}\binom{m}{i}.$$

Therefore, $A \to 0$ as $m \to \infty$.

When, $\frac{m}{2} - m^{\frac{4}{7}} \le i \le \frac{m}{2} + m^{\frac{4}{7}}$, we have $(1 + \lambda_i^H)^n = (1 + o(\frac{1}{m}))^n = 1 + o(1)$. Therefore, $B \approx \sum_{i=\frac{m}{2}-m^{\frac{4}{7}}}^{i=\frac{m}{2}+m^{\frac{4}{7}}} \frac{1}{2^m}\binom{m}{i} \to 1$, as $m \to \infty$.

For $\frac{m}{2}(1 - \epsilon) \le i \le \frac{m}{2} - m^{\frac{4}{7}}$, we get $\lambda_i^H < \lambda^H_{\frac{m}{2}(1-\epsilon)}$. Setting $\epsilon = \frac{1}{m^{1-\theta}}$ (therefore $\epsilon = o(n^{-3})$, $\delta = \sqrt{m}$, and combining with Equation 7, we obtain $\lambda_i^H < \epsilon^3 - \frac{\binom{k}{2}}{m}\epsilon + O(\frac{1}{m^{\frac{3}{2}}})$. Thus,

$$\begin{aligned}(1+\lambda_i^H)^n < e^{m\epsilon^3} e^{-\frac{\binom{k}{2}}{m}\epsilon} &\rightarrow 1\\ m &\rightarrow +\infty \text{ (for } \theta < \frac{2}{3})\end{aligned}$$

We can bound the binomial term in the sum as follows:

$$\begin{aligned}\binom{m}{i} &< \left(\frac{me}{\frac{m}{2}-m^{\frac{4}{7}}}\right)^{\frac{m}{2}-m^{\frac{4}{7}}} = \left(\frac{2e}{1-2m^{\frac{-3}{7}}}\right)^{\frac{n}{2}-m^{\frac{4}{7}}}\\ &< (2e)^{\frac{n}{2}-m^{\frac{4}{7}}}.\end{aligned}$$

Therefore, $\frac{1}{2^m}\binom{m}{i} < \frac{(2\cdot e)^{\frac{n}{2}-m^{\frac{4}{7}}}}{2^m} < (2\cdot e)^{\frac{-m}{14}}$, and

$$\begin{aligned}B &= \sum_{i=\frac{m}{2}-m^{\frac{4}{7}}}^{i=\frac{m}{2}+m^{\frac{4}{7}}} \frac{1}{2^m}\binom{m}{i}(1+\lambda_i^H)^n \rightarrow 0\\ &< 2m^{\frac{4}{7}}(2e)^{\frac{-m}{14}} \rightarrow 0 \text{ as } m \rightarrow +\infty\end{aligned}$$

For the last term

$$\begin{aligned}D = &\sum_{i=\epsilon m}^{\frac{m}{2}(1-\epsilon)} \frac{1}{2^m}\binom{m}{i}(1+\lambda_i^H)^n\\ &+ \sum_{i=\frac{m}{2}(1+\epsilon)}^{(1-\epsilon)m} \frac{1}{2^m}\binom{m}{i}(1+\lambda_i^H)^n \rightarrow 0,\end{aligned}$$

by symmetry we consider the first term. Let $\alpha = \frac{i}{m}$ and $\beta = \frac{n}{m}$. We have [15]

$$\binom{m}{i} < 2^{mH(\frac{i}{m})} = e^{m\log(2)H(\alpha)} = e^{m(-\alpha\log(\alpha)-(1-\alpha)\log(1-\alpha))}.$$

Furthermore, from Lemma 4, we have $\lambda_i^H < (1-\frac{2i}{m})^3 = (1-2\alpha)^3$. Thus,

$$\begin{aligned}&\frac{1}{2^m}\binom{m}{i}(1+\lambda_i^H)^n\\ <&e^{m(-\log 2-\alpha\log(\alpha)-(1-\alpha)\log(1-\alpha)+\beta\log(1+(1-2\alpha)^3))}.\end{aligned}$$

Looking at the exponent in this inequality, we now define a function f which will determine the threshold value c_0. This threshold value is also the expansion threshold used in Theorem 1. Let $f(\alpha,\beta) = -\log 2 - \alpha\log(\alpha) - (1-\alpha)\log(1-\alpha) + \beta\log(1+(1-2\alpha)^3)$. Therefore,

$$\frac{1}{2^m}\binom{m}{i}(1+\lambda_i^H)^n < e^{mf(\alpha,\beta)}.$$

We need to find a value β_0, such that for all α (defined as $\frac{i}{m}$) $\in (\epsilon, 1-\epsilon)$, we guarantee that $f(\alpha,\beta) < 0$. Such value β_0 is found by considering (α_0,β_0) the root of

$$f(\alpha,\beta) = 0 \qquad \frac{\partial f(\alpha,\beta)}{\partial\alpha} = 0.$$

Setting $c > c_0$, we get $\beta < \beta_0$, and therefore $D \rightarrow 0$ as $m \rightarrow +\infty$.

Algorithm 4: $\mathsf{Init}(1^s, \phi, n')$

$K_1^{\text{Super}} \xleftarrow{\$} \{0,1\}^s$;
for $i = 1\ \phi$ **do**
  $\Pi_i.\mathsf{Init}(1^s, n')$;// Store $K_{i,0}$
output $K_{1,0}, \ldots, K_{\phi,0}, K_1^{\text{Super}}$;

Algorithm 5: $\mathsf{addItem}(data_i)$

$\mathsf{pos} = \mathsf{PRG}(K_i^{\text{Super}})$;// Random $pos, 1 \leq pos \leq \phi$
$\Pi_{\mathsf{pos}}.\mathsf{addItem}(data_i, \Pi_{\mathsf{pos}}.DS_i)$;
$K_{i+1}^{\text{Super}} = \mathsf{PRF}_{K_i}(\gamma)$;

Algorithm 6: $\mathsf{listItems}(\Pi_1.DS, \ldots, \Pi_n.DS, K_{1,0}, \ldots, K_{\phi,0}, K_1^{\text{Super}}, n)$

for $i = 1$ **to** ϕ **do**
  $\mathsf{orderedList}_i = \Pi_i.\mathsf{listItems}(\Pi.K_{i,0})$;
for $i = 1$ **to** n **do**
  $\mathsf{pos} = \mathsf{PRG}(K_i^{\text{Super}})$; $K_{i+1}^{\text{Super}} = \mathsf{PRF}_{K_i}(\gamma)$;
  output next item from $\mathsf{orderedList}_{pos}$;

Solving $f(\alpha, \beta) = 0$ for β gives $\beta = (\alpha \cdot \log(\alpha) - \alpha \cdot \log(-\alpha + 1) + \log(2) + \log(-\alpha + 1))/\log(-(2 \cdot \alpha - 1)^3 + 1)$. Using SageMath, we then numerically approximate the minimum β_0 for $0 < \alpha < \frac{1}{2}$ and obtain $\beta_0 = 0.88949$ which means $c_0 = 1.1243$. □

Proof of Theorem 1. For $\delta < \sqrt{n}$, let $c_0 = \frac{1}{\beta_0}$. From Lemma 5, we get: if $c > c_0$ and $m > c \cdot n \implies E(2^d) = A + B + C + D \to 1$ as $m \to \infty$. □

Observe that m is only $\approx 12\%$ larger than n.

6 SUPPORTING LARGE AMOUNTS OF DATA

In practice, the number n of (security critical) data items to be stored can become large, and each data item might have a size ℓ of several hundred bytes. To support large values of n and ℓ, we present two optimizations. First, we present a technique that distributes all n data items into ϕ so called *buckets*. Second, we present a technique to improve performance of Gaussian elimination over large fields.

6.1 Distributing Data Items to Buckets

Instead of one data structure Π to store n data items, we use ϕ smaller data structures $\Pi_1, \ldots, \Pi_\phi$ and distribute all n items to them. Each Π_i is parameterized to store $n' < n$ items and can thus recover only $\delta' \leq \sqrt{n'}$ items. In the following, we call each of these smaller data structures a *bucket*.

During initialization, we initialize all buckets Π_i separately. For then storing the i^{th} item $data_i$, we use yet another key K_i^{Super} to pseudo-randomly choose one of the ϕ data structures Π_i, say Π_*, and call addItem of Π_* to add $data_i$ to Π_*'s internal state. To recover all data items, we call listItems of each Π_i, and use K_0^{Super} to replay all random coins and bring recovered data items into their correct order. We summarize this optimization in algorithms 4 to 6.

Analysis. Assume we configure each Π_i for some n' with $k = 5$ and $c > 1.1243$ as shown previously. By union bound, the probability of successfully decoding *all data in all* ϕ data structures, despite δ' corrupt entries in each of them, remains at $1 - o(1)$ for constant ϕ or $1 - o(\phi)$ which is acceptable for smaller ϕ.

The crucial question is how to choose ϕ and n' to support large values of n. By a standard balls-into-bins argument [34], randomly distributing x balls into y bins results in a maximum number $\mathsf{MBS}(x, y) \leq \frac{x}{y} + \sqrt{\frac{2x \cdot \ln y}{y}}$ of balls in any bin with probability $1 - o(1)$. Thus, we can estimate the lower bound for n' to $n' \geq \mathsf{MBS}(n, \phi)$.

In a crash, the worst case situation is that the whole OS cache contents are only partially written to disk, so δ equals the number of items in the cache. If r is the maximum peak data item rate the system should sustain, and t is, e.g., the cache eviction interval, we can set $\delta \leq r \cdot t \cdot k$ as an upper bound for the sum of the number of all possibly corrupted entries in all data structures. As items are distributed among all Π, we use the same argument from above and have $\delta' = \mathsf{MBS}(r \cdot t \cdot k, \phi)$. For any n' we select, $\sqrt{n'} \geq \delta'$ must hold.

Using r, t, k, and n, this allows choosing ϕ and n' such that $\delta' \leq \sqrt{n'}$. We present and benchmark several combinations in our evaluation in Section 7.

6.2 Smaller Field Sizes

Our listItems in Algorithm 3 solves a system of linear equations using Gaussian elimination. However, vector $\vec{c}$ is a vector over field $(GF(2^{\ell+2\cdot s}))^n$. For many real-world values of plaintext length ℓ, field sizes become very large. For example, for syslog events of maximum length 1024 byte [25], 128 bit random counters, and 256 bit HMAC-SHA2, operations must be in $GF(2^{8576})$. Gaussian elimination over large fields is extremely slow in practice even when using modern, optimized computer algebra systems. To mitigate the problem, we exploit a special property of our addItem algorithm. Observe that XOR operations during addItem and M's coefficients of either 0 or 1 make M a (sparse) matrix over $GF(2)$. Thus, we use the following tweak to speed up solving.

Instead of solving $M \cdot \vec{x} = \vec{v}$ over $(GF(2^{\ell+2\cdot s}))^n$, we compute a reduced row echelon form E of M together with a $m \times m$ transformation matrix T such that $T \cdot M = E$. As all operations are over $GF(2)$, computations of E and T are fast, see the evaluation in Section 7. As $M \cdot \vec{x} = \vec{v}$, we multiply with T from the left and get $T \cdot M \cdot \vec{x} = E \cdot \vec{x} = T \cdot \vec{v}$. So, we convert T from $GF(2)$ into $(GF(2^{\ell+2\cdot s}))^n$ and multiply T with $\vec{v}$ to get $\vec{x}$. T's conversion is cheap, and instead of cubic complexity for Gaussian elimination over a large field, multiplying T with $\vec{v}$ has only quadratic complexity.

As a result, Gaussian elimination becomes significantly faster. To illustrate, computing the rank of a 500×500 random matrix over $GF(2)$ is roughly 4000× faster than computing the rank of the exact same matrix converted to $GF(2^{8576})$ on a 2.2 GHz mobile Intel Skylake i7 CPU (see Section 7 for more experiments). In conclusion, this optimization results in a significant speed-up of orders of magnitude.

7 EXPERIMENTAL ANALYSIS

To back up our theoretical claims, we have also performed a practical analysis of our coding technique. The goal of this analysis is twofold. First, we estimate and give confidence in the probability that listItems recovers all data for a concrete choice of parameters k, c, n, and δ following our theoretical prediction in Section 5.2. Second, we show that even the more involving decoding is concretely practical by measuring its runtime for logs of different size up to millions of log items ($n = 2^{23} \approx 8.4$ million). For such large logs we rely on our multi-bucket technique of Section 6.

We have implemented our coding and decoding techniques in *C*, and the implementation is available for download [7].

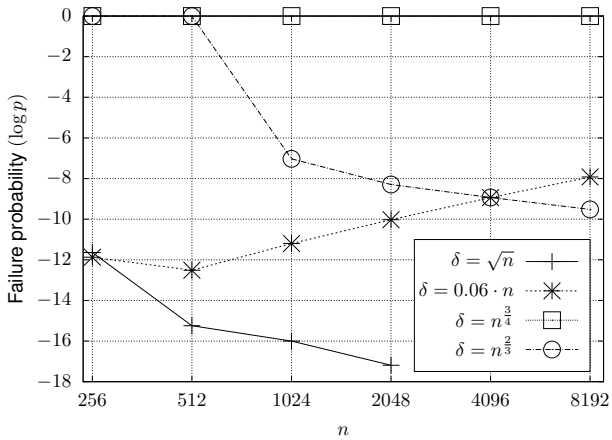

Fig. 3. Failure probability of listItems

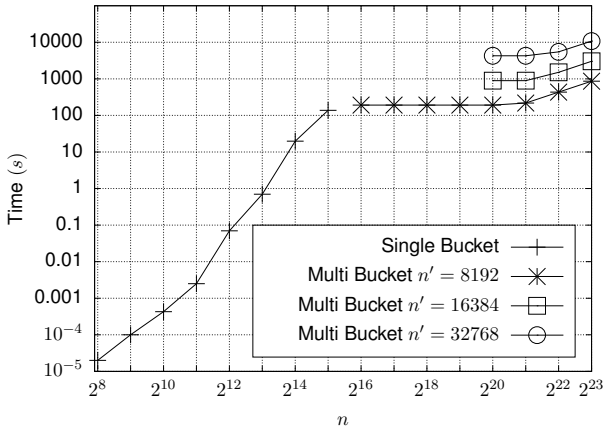

Fig. 4. Runtime of listItems

7.1 Failure Probability

Our implementation first builds a random binary $(m = c \cdot n) \times n$ matrix M where each column has weight k. Thereby, we simulate encoding (and addItem) for a total of n data items. We also simulate adversarial behavior and crashes: δ rows are randomly chosen and removed from M by setting all coefficients to 0. To simulate decoding (and listItems), the implementation computes Gaussian elimination to convert M into reduced row echelon form and derives M's rank.

As our (standard) Gaussian elimination is asymptotically expensive with $O(n^3)$ complexity, the implementation features two performance improvements. First, M is a binary matrix and Gauss' operations are only XOR of rows. We use Intel's AVX2 SIMD instructions, store rows in 256 bit pieces, and XOR each pair of pieces with only one CPU instruction. Our second performance improvement integrates OpenMP parallel processing for the inner loop of Gaussian elimination. Especially for larger n, parallelization gives performance improvements. We also stress that, in our case of sparse matrices M, Gaussian elimination allows for significant performance improvements [44] which we leave to future work.

We run our experiments with $k = 5$ and $c = 1.1244$, as suggested by Section 5.2, for $n \in \{256, 512, 1024, 2048, 4096, 8192\}$ and $\delta = \sqrt{n}$. By that, we will indicate that our theory of recovering up to $\sqrt{n}$ lost or corrupted entries in M can

Table 2. Configuration, $k = 5$, $c = 1.1244$, $t = 30s$.

n	n'	ϕ	$r\,(\frac{\text{logs}}{s})$	Total Space Overhead
2^{20}	8192	236	112	×2.07
	16384	156	100	×2.74
	32768	104	100	×3.65
2^{21}	8192	266	169	×1.17
	16384	156	100	×1.37
	32768	104	100	×1.83
2^{22}	8192	533	220	×1.17
	16384	263	166	×1.16
	32768	131	125	×1.15
2^{23}	8192	1068	432	×1.17
	16384	527	329	×1.16
	32768	261	246	×1.15

be recovered with high probability. We additionally run experiments with $\delta = n^{\frac{3}{4}}$, $\delta = n^{\frac{2}{3}}$, and even $\delta = 0.06 \cdot n$, i.e., a small, but constant fraction of n. The idea here is to show that, as predicted, our coding cannot recover from values of δ significantly larger than $\sqrt{n}$ with high probability. For each combination of parameters, we perform 2^{20} repetitions and compute $p = \frac{\#(M \text{ has rank} < n)}{2^{20}}$.

Figure 3 shows the outcome of these experiments. The x-axis (log scaled) shows the number of data items, and the y-axis shows $\log p$. For $\delta = \sqrt{n}$ and both $n = 4096$ and $n = 8192$, we have $p = 0$, i.e., all 2^{20} random matrices M had full rank n. So, starting with relatively low values of n (e.g., $n = 4096$), the system does not experience any failures, backing up our formal analysis of setting $\delta < \sqrt{n}$ to decode with high probability. In contrast, increasing δ significantly beyond $\sqrt{n}$ jeopardizes decodability. For $\delta = n^{\frac{3}{4}}$, all M had rank less than n, and for both $\delta = n^{\frac{2}{3}}$ and $\delta = 0.06 \cdot n$ failure probability is significant. These results are consistent with our theoretical asymptotic bounds. For $\delta \geq n^{\frac{3}{4}}$ the recovery fails with high probability, supporting the claim of near-optimality. We leave it as an open question to derive a tighter bound on the recoverability of the proposed scheme.

We also stress that the secure crash recovery setup is different from traditional security setups: in each run, the adversary only gets *one* chance to tamper with the system log. So, in 2^{20} experiments, not a single time was the adversary able to delete δ entries that the system could not recover.

Failure of cryptographic primitives. We also emphasize that the probability of failure of the cryptographic primitives we employ (AES and HMAC) is negligible in their security parameter s. In our evaluation, we set $s = 256$, i.e., we use AES with standard 256 Bit keys and HMAC-SHA256.

7.2 Runtime of listItems

Figure 4 shows listItems runtime on a 16 Core Xeon E5-2630 2.2 GHz computer. From $n = 256$ to $n = 8192$, we decode by internally computing Gaussian ("Single Bucket") elimination as above. The runtime scales closely to the anticipated cubic complexity. Note that total runtime of decoding, including decryption of log entries, HMAC verification etc. is dominated by Gaussian elimination. For example, with $n = 32768$ log entries, each of size 1KByte, Gaussian elimination takes 91s, but AES decryption and HMAC integrity verification a total of only 1s.

Benchmarks for large logs. To support large amounts of data, i.e., values $n > 8192$, we employ our "Multi-Bucket" optimization technique from Section 6.1 and set n' to one of 8192, 16384 or 32768. Following $\delta' = \text{MBS}(rtk, \phi) \leq \sqrt{n'}$

Table 3. High-level comparison to related work (techniques without crash recovery vs. techniques with crash recovery). Naïve coding: a strawman construction that would combine a standard approach for forward integrity (e.g., [4, 5]) with conventional coding, ϕ : number of buckets (Section 6)

Protocol	Forward Integrity	Public Verifiability	Efficient Non-Membership	Crash Tolerance	Crash Recovery	Complexities addItem	Complexities listItems
[4, 5, 8, 20, 27–29, 37]	✓	—	—	—	—	$O(1)$	$O(n)$
[27, 46, 47]	✓	✓	—	—	—	$O(1)$	$O(n)$
[33]	✓	✓	✓	—	—	$O(\log n)$	$O(n \log n)$
[3, 6, 31]	✓	—	—	✓	—	$O(1)$	$O(n)$
Naïve coding	✓	—	—	✓	✓	$O(n)$	$O(n)$
Ours (single bucket)	✓	—	—	✓	✓	$O(1)$	$O(n^{2.4})$
Ours (multi bucket)	✓	—	—	✓	✓	$O(1)$	$O(\phi \cdot (\frac{n}{\phi})^{2.4})$

and $\mathsf{MBS}(n, \phi) \leq n'$, we set ϕ (Section 6.1) such that the peak rate of logs per second r is at least $100\frac{\text{logs}}{s}$. For cache eviction interval t, we use the Linux standard kernel value of $t = 30$s [24]. Table 2 summarizes configuration parameters used. A value of $n = 2^{23}$ corresponds to a log with 8 million entries, each of 1024 Byte length and a total size of 8 GByte, largely sufficient to record the security logs of a wide variety of practical systems. Moreover, since the buckets are processed independently, computation scales *linearly* in the number of buckets, and one could realize a parallelization that decodes each bucket separately. For example, a parallel implementation would decode a 16 GByte log on a computer with 32 Cores in the same time as a our implementation decodes the 8 GByte, $n = 2^{23}$ log. In the real world, log file sizes vary depending on the concrete scenario. They range from a few KByte on a personal computer to many GByte on server systems. Yet, we point out that our techniques are primarily targeting security critical logs. They are often significantly smaller than logs capturing the events of a whole system. As demonstrated by our experiments for logs of n up to 2^{23} and the scalability due to parallelization of the multi-bucket technique, we conclude that listItems is practical even for large security logs. Observe that our techniques obviously support larger logs at an increase in runtime.

Finally, we also point out that constructions exist to reduce the asymptotic complexity of Gaussian elimination from n^3 down to $n^{2.4}$ by building on Coppersmith and Winograd [11] multiplication and the Strassen [42] transformation. We leave applying such optimizations for future work.

8 RELATED WORK

Secure logging with forward integrity has received some attention, see [4, 5, 8, 18–20, 27–29, 31, 33, 37, 46, 47] for an overview. However, coping with crash attacks was severely limited so far [3, 6, 31]. An analyst could only distinguish whether an inconsistency in a log file is due to adversarial modifications or to a real-world crash. In contrast, the goal of this paper is to treat data lost in a real crash or modified during a crash attack by using a special encoding of logged data. We treat lost and modified data in the same way and recover up to a configurable amount of δ lost or modified data items. So, we do not just distinguish between a real crash and a crash attack as previous work, but either recover from a real crash, neutralize adversarial modifications or detect the adversary.

Directly comparable work. We summarize related work and compare it to our techniques in Table 3. We note that other works on secure logging feature advanced security properties such as public verifiability [27, 46, 47] or efficient tests for non-membership [33] of a log event in a file. Similarly, other works also target sophisticated security properties such as *excerpt* verification of subsequences of log events [18]. While these are important security properties, we envision scenarios where crash recovery might be more important, and these properties are therefore non-goals (e.g., systems with

high chances of crashes, simple embedded systems where public verifiability or non-membership tests do not matter). Still, we stress that providing such additional security properties are worth future investigations. In conclusion, one might understand our solution as a first step to crash recovery and a direct extension of schemes like [4, 5, 8, 20, 27–29, 37]. We leave the additional combination with, e.g., public verifiability and efficient non-membership tests open to subsequent work.

8.1 Logging based on stronger assumptions

An interesting line of research has recently addressed securing logging using trusted execution environments (TEEs) like Intel's SGX [21, 30]. Other works have investigated secure logging given a trusted hypervisor that can observe and log security critical events in its VMs [22, 38]. Given the availability of TEEs or trusted software components such as hypervisors, previous works provide stronger security guarantees than the ones targeted in this paper. Instead of *tamper evidence*, they achieve *tamper-proofness*, i.e., it becomes impossible for an adversary to modify or delete security-critical log data.

While TEEs or trusted software components provide tamper-proofness, we argue that weaker tamper evidence as targeted in this and other works [3–6, 8, 20, 27, 27–29, 31, 33, 37, 46, 47] is still helpful: in some scenarios, TEEs might be too expensive, complex to use (cloud scenarios) or simply not available. Also, trusting software or hardware components might also be dangerous as demonstrated by various security flaws in the Xen hypervisor [45] (and others) or Intel's TEE SGX, see van Schaik et al. [43] for an overview.

Moreover, while TEEs and trusted software components allow for elegant solutions to provide forward-integrity and tamper-proofness, they do not solve the problem of crashes which is the focus of this work. Without the results of this paper, one would need to employ additional special, dedicated hardware devices [1] to circumvent the problem of crashes.

8.2 Data Structures with Redundancy

There exists previous work on data structures with redundancy which has served as a motivation for this work. Goodrich and Mitzenmacher [17] and Pontarelli et al. [32] store data in a similar fashion as our addItem operation, but later recover by only using a *peeling* mechanism. That is, they check a table of XORs of data item replicas for cells containing only one replica. As long as they find such a cell, they remove the replica from all other cells containing the replica. While peeling (and its analysis) is simple and elegant, it limits the performance to recover all data. In contrast, our rather complex decoding and its analysis show that the encoding allows for high decodability guarantees. Specifically, Goodrich and Mitzenmacher [17] can recover only from a fixed, constant number of lost or modified data items, independently of n, while we support up to $\sqrt{n}$ lost or modified items. In addition, our storage overhead is significantly less: for a similar configuration where a data item is written into $k = 5$ cells in a table, Goodrich and Mitzenmacher [17] require an additional 43% space overhead while we need only 12%. While theoretically possible, none of the previous works provides forward integrity or forward confidentiality as this paper does.

8.3 Conventional Coding

At the heart of our approach is a random linear code. Error and erasure correction codes have been extensively studied since the establishment of information theory in 1948 and the proof of existence of capacity achieving codes for a variety of channels [12, 39]. Our proposed code bears some similarities to LDPC codes and LT codes. However, it is uniquely restricted by the secure logging requirements, namely that every log data item (symbol) is encoded at once (in an online

manner), using a constant number of operations, without maintaining information about the symbols involved in the encoding (to provide forward integrity).

Well studied codes such as Reed-Solomon, LDPC, and Fountain codes (LT, Raptor, Tornado) are very efficient in terms of rates and erasure correction ability [23, 26, 41]. Yet, they have constraints that make them unusable in the context of secure logging. In particular, such codes require operations over a non-constant number of symbols for each input symbol, or access to previously coded symbols. For instance, in Reed-Solomon codes, each uncoded input symbol (log data item) contributes to all coded symbols. The uncoded symbols are the coefficients of a polynomial evaluated over the powers of a primitive root of unity [23]. The evaluation results are the coded symbols. Therefore, changing a single uncoded symbol (log entry) would result in changing all the coded symbols. LDPC codes have sparse parity matrices, but their generator matrices are not sparse [16, 40]. Therefore, changing a single uncoded symbol requires the update of a large number of coded symbols.

The closest type of codes applicable to this context are Fountain codes, e.g., LT-codes or Raptor codes. While these codes have good erasures-correction performance in typical communications and storage contexts, they cannot be used in this context as they impose requirements on the distribution of the encoded symbols degree (in the bi-partite graph representation) such as the Soliton distribution for LT-codes [26], or rely on the explicit concatenation of an inner and outer codes [41]. The first conflicts with the requirement of a constant number of operations per encoded symbol. Furthermore, the forward-security requirements prevent enforcement of such degree distributions, as, at any instant of time, the system should not have knowledge of any information about previously encoded symbols (in particular their degree). We acknowledge that, in return, our proposed coding scheme is not suitable for conventional communication and storage scenarios, as secure logging requirements limit its data recovery performance.

9 CONCLUSION

We have presented a new data structure Π together with several new mechanisms combining forward integrity and confidentiality with data recovery. At the core of our techniques lies an efficient, customizable coding scheme with high decodability guarantees even when a large number of data items are lost or maliciously modified. This coding scheme is online, requires a constant number of operations per input symbol, and therefore enables the integration of forward security mechanisms. Our formal analysis and practical experiments show that for any number of log items, a space overhead of only 12% and a computational overhead of a factor of 5 suffices to decode and recover all log items with high probability. The coding scheme is of independent interest on its own. While secure and robust audit data storage is a prime application for our techniques, one can envision other applications. Whenever an adversary threatens to compromise a system, tamper with data, and tries hiding traces to avoid detection, our techniques will be useful.

REFERENCES

[1] A. Ahmad, S. Lee, and M. Peinado. HARDLOG: practical tamper-proof system auditing using a novel audit device. In *43rd IEEE Symposium on Security and Privacy, SP 2022, San Francisco, CA, USA, May 22-26, 2022*, pages 1791–1807. IEEE, 2022.

[2] Y. Aumann and Y. Lindell. Security against covert adversaries: Efficient protocols for realistic adversaries. *Journal of Cryptology*, 23(2):281–343, 2010. ISSN 0933-2790.

[3] S. Avizheh, R. Safavi-Naini, and S. Li. Secure Logging with Security against Adaptive Crash Attack. In *International Symposium on Foundations & Practice of Security*, Toulouse, France, 2019. https://arxiv.org/abs/1910.14169.

[4] M. Bellare and B.S. Yee. Forward integrity for secure audit logs. Technical report, UC San Diego, 1997.

[5] M. Bellare and B.S. Yee. Forward-Security in Private-Key Cryptography. In *Topics in Cryptology - CT-RSA 2003, The Cryptographers' Track at the RSA Conference 2003, San Francisco, CA, USA, April 13-17, 2003, Proceedings*, pages 1–18, 2003.

[6] E.-O. Blass and G. Noubir. Secure Logging with Crash Tolerance. In *Conference on Communications and Network Security*, pages 1–10, Las Vegas, USA, 2017.

[7] E.-O. Blass and G. Noubir. Source code for experiments, 2022. https://github.com/dalmayr777/secure-logging.

[8] K.D. Bowers, C. Hart, A. Juels, and N. Triandopoulos. PillarBox: Combating Next-Generation Malware with Fast Forward-Secure Logging. In *RAID*, pages 46–67, 2014.

[9] N.J. Calkin. Dependent sets of constant weight vectors in $GF(q)$. *Random Struct. Algorithms*, 9(1-2):49–53, 1996.

[10] N.J. Calkin. Dependent Sets of Constant Weight Binary Vectors. *Combinatorics, Probability & Computing*, 6(3):263–271, 1997.

[11] D. Coppersmith and S. Winograd. Matrix multiplication via arithmetic progressions. *J. Symb. Comput.*, 9(3):251–280, 1990.

[12] T.M. Cover and J.A. Thomas. *Elements of Information Theory (Wiley Series in Telecommunications and Signal Processing)*. Wiley-Interscience, 2006.

[13] M. Dietzfelbinger and R. Pagh. Succinct Data Structures for Retrieval and Approximate Membership (Extended Abstract). In *ICALP*, pages 385–396, 2008.

[14] P. Erdős and A. Rényi. On the Evolution of Random Graphs. In *Publication of the Mathematical Institute of the Hungarian Academy of Sciences*, pages 17–61, 1960.

[15] J. Flum and M. Grohe. *Parameterized Complexity Theory*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2006.

[16] R.G. Gallager. Low-density parity-check codes. *IRE Trans. Information Theory*, 8(1):21–28, 1962.

[17] M.T. Goodrich and M. Mitzenmacher. Invertible Bloom Lookup Tables. In *Allerton Conference on Communication, Control, and Computing*, pages 792–799, Monticello, USA, 2011.

[18] G. Hartung. Secure audit logs with verifiable excerpts. In *CT-RSA*, volume 9610 of *LNCS*, pages 183–199, 2016.

[19] V. T. Hoang, C. Wu, and X. Yuan. Faster Yet Safer: Logging System Via Fixed-Key Blockcipher. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 2389–2406, Boston, MA, 2022. USENIX Association. ISBN 978-1-939133-31-1.

[20] J.E. Holt. Logcrypt: forward security and public verification for secure audit logs. In *Australasian Symposium on Grid Computing and e-Research*, pages 203–211, 2006.

[21] V. Karande, E. Bauman, Z. Lin, and L. Khan. SGX-Log: Securing System Logs With SGX. In *AsiaCCS*, pages 19–30. ACM, 2017.

[22] Samuel T. King and Peter M. Chen. Backtracking intrusions. In *19th ACM Symposium on Operating Systems Principles, Bolton Landing, NY, USA*, pages 223–236, 2003.

[23] S. Lin and D.J. Costello. *Error Control Coding, Second Edition*. Prentice-Hall, Inc., USA, 2004. ISBN 0130426725.

[24] Linux Kernel Documentation. `/proc/sys/vm/dirty_expire_centisecs`, 2020. Standard value is 30 sec on kernel 5.8, 64 bit, https://www.kernel.org/doc/Documentation/sysctl/vm.txt.

[25] Chris M. Lonvick. The BSD Syslog Protocol. RFC 3164, August 2001. URL https://rfc-opteditor.org/rfc/rfc3164.txt.

[26] M. Luby. LT Codes. In *IEEE Annual Symposium on Foundations of Computer Science*, 2002.

[27] D. Ma and G. Tsudik. A New Approach to Secure Logging. *ACM Transactions on Storage*, 5(1), 2009. ISSN: 1553-3077.

[28] G.A. Marson and B. Poettering. Practical Secure Logging: Seekable Sequential Key Generators. In *ESORICS*, pages 111–128, 2013.

[29] G.A. Marson and B. Poettering. Even More Practical Secure Logging: Tree-Based Seekable Sequential Key Generators. In *ESORICS*, pages 37–54, 2014.

[30] R. Paccagnella, P. Datta, W. Ul Hassan, A. Bates, C.W. Fletcher, A. Miller, and D. Tian. Custos: Practical Tamper-Evident Auditing of Operating Systems Using Trusted Execution. In *NDSS*. The Internet Society, 2020.

[31] R. Paccagnella, K. Liao, D. Tian, and A. Bates. Logging to the danger zone: Race condition attacks and defenses on system audit frameworks. In *Conference on Computer and Communications Security*, pages 1551–1574, 2020.

[32] S. Pontarelli, P. Reviriego, and M. Mitzenmacher. Improving the performance of Invertible Bloom Lookup Tables. *Inf. Process. Lett.*, 114(4):185–191, 2014.

[33] T. Pulls and R. Peeters. Balloon: A forward-secure append-only persistent authenticated data structure. In *ESORICS*, volume 9327 of *LNCS*, pages 622–641, 2015.

[34] M. Raab and A. Steger. Balls into Bins – A Simple and Tight Analysis. In *RANDOM'98*, volume 1518 of *LNCS*, pages 159–170, 1998.

[35] T. J. Richardson and R. L. Urbanke. Efficient encoding of low-density parity-check codes. *IEEE Transactions on Information Theory*, 47(2):638–656, Feb 2001.

[36] P. Rogaway. Nonce-Based Symmetric Encryption. In *Proceedings of FSE*, pages 348–359, Delhi, India, 2004. ISBN 3-540-22171-9.

[37] B. Schneier and J. Kelsey. Secure audit logs to support computer forensics. *ACM Transactions on Information and System Security*, 2(2):159–176, 1999.

[38] K. F. Seiden and J. P. Melanson. The Auditing Facility for a VMM Security Kernel. In *IEEE Symposium on Security and Privacy, Oakland, California, USA*, pages 262–277. IEEE Computer Society, 1990.

[39] C.E. Shannon. A mathematical theory of communication. *The Bell System Technical Journal*, 27(3):379–423, July 1948.

[40] A. Shokrollahi. *LDPC Codes: An Introduction*, volume 23 of *Coding, Cryptography and Combinatorics*, pages 85–110. Birkhäuser, 2004. ISBN 978-3-0348-9602-3.

[41] A. Shokrollahi. Raptor codes. *IEEE Transactions on Information Theory*, 52(6):2551–2567, 2006.

[42] V. Strassen. Gaussian elimination is not optimal. *Numer. Math.*, 13(4):354–356, 1969. ISSN 0029-599X.

[43] Stephan van Schaik, Alex Seto, Thomas Yurek, Adam Batori, Bader AlBassam, Christina Garman, Daniel Genkin, Andrew Miller, Eyal Ronen, and Yuval Yarom. SoK: SGX.Fail: How stuff get eXposed. 2022.

[44] D. Wiedemann. Solving sparse linear equations over finite fields. *IEEE Transactions on Information Theory*, 32(1):54–62, January 1986. ISSN 0018-9448.

[45] XEN. (Incomplete) list of security flaws in XEN allowing to break out of VM, 2023. https://xenbits.xen.org/xsa/advisory-148.html, https://xenbits.xen.org/xsa/advisory-182.html, https://xenbits.xen.org/xsa/advisory-212.html, https://xenbits.xen.org/xsa/advisory-213.html, https://xenbits.xen.org/xsa/advisory-214.html, https://xenbits.xen.org/xsa/advisory-215.html.

[46] A.A. Yavuz, P. Ning, and M.K. Reiter. BAF and FI-BAF: Efficient and Publicly Verifiable Cryptographic Schemes for Secure Logging in Resource-Constrained Systems. *Transactions on Information System Security*, 15(2):9, 2012. ISSN 1094-9224.

[47] A.A. Yavuz, P. Ning, and M.K. Reiter. Efficient, compromise resilient and append-only cryptographic schemes for secure audit logging. In *Financial Cryptography and Data Security*, volume 7397 of *LNCS*, pages 148–163, 2012.