CONSTRUCTING A CONFIDENTIAL, AUTHENTICATED, FORWARD SECURE AND OFFLINE LOGGING SCHEME

P. C. J. VAN DER VEEKEN

January 2018

TU Delft

Constructing a Confidential, Authenticated, Forward Secure and Offline Logging Scheme

by

P. C. J. van der Veeken

in partial fulfillment of the requirements for the degree of

Master of Science
in Computer Science

at the Delft University of Technology,
to be defended publicly on Thursday January 25, 2018 at 2:00 PM.

Student number:	4095812	
Project duration:	April 24, 2017 – January 25, 2018	
Thesis committee:	Assoc. Prof. Dr. Ir. J.C.A. van der Lubbe,	TU Delft
	Assoc. Prof. Dr. Ir. M. de Weerdt,	TU Delft
	Assist. Prof. Dr. Z. Erkin,	TU Delft
	Ir. H. P. Westen	Fox-IT
Supervisors:	Assist. Prof. Dr. Z. Erkin,	
	Ir. H. P. Westen	

This thesis is confidential and cannot be made public until January 25, 2018.

ABSTRACT

In IT systems, a logfile provides administrators with an audit trail which can be used to review a system's activities and a way to discover and diagnose problems which have occurred within that system. When an attacker penetrates an IT system, commonly one of their first actions is tampering with the logs, so that they can hide their malicious activities [CP10]. For most types of systems a suitable secure logging solution exists. These solutions prevent an adversary from tampering with the system logs, or make it infeasible to do so undetectably. However, the solutions that exist for devices with limited computing power, "resource-constrained" devices, are all unsuited for long-term unsupervised deployment. In this scenario a device is deployed in a hostile environment for prolonged periods of time, during which it cannot communicate or otherwise interact with another party. Existing solutions for this scenario are either, not secure against all known attacks on secure logging schemes, make assumptions that are not realistic given the aforementioned scenario, or fail to account for real-world constraints that these devices and scenario impose on the capabilities of a secure logging scheme.

In this thesis we present two secure logging schemes called *Immutable Forward Linked and Sealed Logging* (IFLS) and *Pseudorandom Indexed Forward Linked Logging* (PIFL). Both schemes allow for tamper-resistant logging on low-powered devices while at the same time requiring only initial interaction with an external party. Furthermore we present a novel and efficient way of establishing an immutable link between consecutive log entries, which grants protection against most attacks on secure logging schemes. We also detail two methods to shield the last log entry, so that Truncation Attacks and Crash Attacks [BN17] are mitigated. The combination of these techniques in IFLS and PIFL results in two schemes which are fully tamper-resistant. We additionally find that PIFL's pseudorandom indexing of log entries makes this scheme well-suited for use on flash storage, a storage medium that is ubiquitous in resource-constrained devices. Lastly, we confirm the real-world feasibility of our work by implementing and practically evaluating IFLS in the Rust programming language.

"The true delight is in the finding out rather than in the knowing."

— Isaac Asimov [Asi20]

"One day I will find the right words, and they will be simple."

— Jack Kerouac [KD06]

ACKNOWLEDGMENTS

This thesis would not have been possible without the unwaivering support of my family and friends. You made sure I came up for air if I was submerged for too long and kept me grounded when I was in danger of drifting off.

Special thanks go out to both of my supervisors, Pepijn and Zeki. Pepijn, our daily meetings were excellent opportunities for me to float a lot of stupid ideas, and occasionally slightly less stupid ideas. Something which I am deeply grateful for. Zeki, your relentless pushing kept me ambitious and motivated in moments where I would have become complacent otherwise. Furthermore, the warm and friendly environment you cultivate within your research group is greatly valued not just by me, but indeed by all of your students.

I would also like to express my gratitude to all my fellow interns, Master- and PhD students with whom I had the pleasure to interact with over the course of the last nine months. You were there to celebrate the highs, to make the lows seem less low, or to sometimes just forget about research in favor of cake and slightly off-color jokes.

Lastly, I want to thank Fox-IT for providing me with a, friendly and positive environment in which I could peacefully work on my thesis. In particular I would like to thank Hielke, Joachim and Steffan for helping me crack some (at least to me) very tough cryptographic problems.

I will close on the note that I will forever treasure the connections I have made, the things I have learned and even the things I could not learn. All of these things I have mentioned, and probably too those I have forgot to mention, have helped me in my transition from clueless Master student to an ever so slightly less clueless Engineer.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACRONYMS

IFLS Immutable Forward Linked and Sealed

PIFL Pseudorandom Indexed Forward Linked logging

PKC Public Key Cryptography

XMSS Extended Merkle Signature Scheme

OTS One-Time Signature

TSS Time-Stamping Service

MAC Message Authentication Code

HMAC Hashed Message Authentication Code

IBS Identity Based Signature

GDH Gap Diffie-Hellman (groups)

CDHP Computational Diffie-Hellman Problem

DDHP Decisional Diffie-Hellman Problem

BAF Blind Aggregate Forward

HASAFSS Hash-Based Sequential Aggregate Forward Secure Signature Scheme

ECC Elliptic Curve Cryptography

TRE Timed-Release Encryption

IDS Intrusion Detection System

UBER Uncorrectable Bit Error Rate

PRF Pseudorandom Function

EU-CMA Existential Under Chosen Message Attacks

IND-CCA Indistinguishable under Chosen Ciphertext Attacks

AES Advanced Encryption Standard

TTP Trusted Third Party

1 INTRODUCTION

Log keeping is the act of recording relevant information and events about a process in a log file [RP01]. In the computing infrastructure domain log files are a vital part of any system; They provide administrators with an audit trail which can be used to review the system's activities and a way to discover and diagnose problems which have occurred within the system. For systems which do not operate in a high security context, such as, refrigerators or internet connected light bulbs, a simple log file containing the logged messages in chronological order is sufficient, should their actions ever need to be reviewed. However, for systems which operate in high-risk environments, such as military sensors or a large-scale cloud infrastructure, one has to account for malicious entities trying to hide their activities by tampering with the log [KS06]. In these scenarios a secure logging scheme is needed which can guarantee the integrity and authenticity of the log's contents.

The function and utility of a secure logging scheme can best be illustrated with an example; DARPA's LANdroids [Esh07] are little caterpillar-tracked robots which are deployed behind enemy lines to collect military intelligence. It is likely that these measurements are stored in a log-like data structure, where each measurement is a new entry in the log. In the event that one of these robots is taken over by an adversary we would have to assume the worst possible scenario, which is to say the adversary controls the robot completely. If we take a moment to reflect on this scenario, we come to the conclusion that no amount of security measures will protect against the adversary entering false information into the log after they have assumed control [SK98]. However, we make the assumption that, provided that the pre-compromise log entries remain intact and unaltered, it is possible to detect the intrusion after the fact with overwhelming probability. In this last condition lies the challenge for secure logging schemes: ensuring the authenticity and integrity of the pre-compromise part of the log file.

Many different schemes have been proposed for creating secure audit logs. However all of them either require some form of online communication [Acc13] [AKV03] [CW09] [Dow+16] [Hol06] [SK98] [Wat+04] [YNR12] [YN12] or are vulnerable to one or more attacks [MT09] [SK98] [YNR12] [YN12] [Hol06]. Most notably, all of these papers are vulnerable to the Crash attack [BN17]. Furthermore only a few of these schemes were designed specifically for platforms with limited computing power [YN12] [MT09] [CW09] [BN17]. Given the need for secure logging systems, it is essential to have an efficient and secure protocol. In this thesis we design a secure logging scheme which is both efficient and secure and which offers significant improvements over existing solutions.

1.1 SECURE LOGGING SCHEMES

A secure audit log is defined by two properties; **(i) Tamper resistance** and **(ii) Verifiability**. Firstly, a secure audit log should be tamper resistant. That is, it must guarantee that the logger and only the logger can create valid entries (authentic-

ity). Additionally, once an entry has been created it must not be possible to alter it (forward security). However, as mentioned above, once an attacker has compromised a system one cannot prevent them from controlling what data is entered into the log. Furthermore, one can also not prevent the attacker from deleting log entries that have not been transferred to an external system [SK98]. The utility of secure logging schemes in this case lies in not allowing an attacker to perform these actions undetectably to any party inspecting the log afterwards.

Secondly, a log should be verifiable. As mentioned earlier, an attacker must not be able to undetectably alter any log entries created before compromise. To this end there needs to be an external party inspecting an audit log to verify whether all entries are present and untampered with. The external party can come in many forms such as, a trusted computer [SK98], a dedicated verifier [Acc13] or any interested party in the case of publicly verifiable audit log [Adi08].

In order for an audit log to be verifiable, it needs to contain two types of information. Firstly, it needs to provide data with which each individual entry's authenticity can be verified. The reason for this requirement is that in case that the log is damaged or some entries are deleted, it should still be possible to recover information from undamaged parts of the log. Secondly, the entries need to be connected to one another in such a way that the order of the entries can be verified and deletions can be detected. The combination of these two information types should make it infeasible for an attacker to undetectably tamper with the entire pre-compromise part of the audit log.

1.2 APPLICATION SCENARIO AND DEVICE LIFE CYCLE

The term "resource constrained" is used often throughout this thesis. Whenever this term is used, we mean a device which only has limited computing power and memory at its disposal. The consequence hereof is that with respect to cryptographic operations certain limitations are implicitly imposed on the capabilities of $\mathcal{U}$. Concretely, this means that most asymmetric cryptographic primitives cannot be used by $\mathcal{U}$, because these typically require much more computational power than symmetric ones [EKK14].

The scenario we base our research around is based on a typical military deployment process of (semi-)autonomous devices. In this scenario a device's life cycle starts at a trusted and secure location, where it is prepared for deployment. After preparations have finished, the device is deployed in a hostile or otherwise unsafe territory. Here the device is expected to operate for prolonged periods of time during which it is not overseen nor contacted by the party which deployed it. If the device survives deployment or is not otherwise lost, it will eventually be collected by the aforementioned party.

The above process is the basis for the application scenario we envision. In this scenario we have a resource constrained, untrusted logger $\mathcal{U}$. $\mathcal{U}$ is expected to operate autonomously for indefinite periods of time under harsh conditions (e. g. can experience power failures, overheating, etc.). Untrusted in this context means $\mathcal{U}$ is not sufficiently tamper-resistant or physically secure to prevent an attacker $\mathcal{A}$ from taking it over. Over the duration of its deployment it cannot be assumed that $\mathcal{U}$ will

ever be able to communicate with a third-party. However, it can be assumed that the device will be inspected by a trusted party $\mathcal{T}$ if it were to stop functioning correctly, or it has detected an intrusion. Furthermore on pre-deployment $\mathcal{T}$ has the ability to safely perform operations such as key sharing, initializing generating public/private key pairs and setting security parameters.

In the military branch it is often the case that equipment is developed and produced by external contractors. If the said equipment malfunctions in this scenario, the military might not have the necessary expertise to diagnose the problem themselves. If this is the case, the device needs to be sent to the external contractor for review. From the military point of view however, this poses a problem. The device might contain sensitive information which should not be viewed by external parties. Furthermore, the military might not completely trust the contractor to not somehow modify, delete, or otherwise tamper with data on the device. A solution to this issue is *selective verifiability and disclosure*. This means that verification and disclosure of the data on the device is distributed over separate parties. The verifying party $\mathcal{V}$ can only verify the integrity of the data and the inspecting party $\mathcal{I}$ can only read the data on the device. Both of these parties are subordinate to $\mathcal{T}$, meaning that only $\mathcal{T}$ can give them access to (parts) of the data on $\mathcal{U}$.

1.3 SECURITY GUARANTEES

If $\mathcal{A}$ compromises $\mathcal{U}$ at a point of time t no guarantees can be made about log entries recorded after t. Therefore we are forced to assume that all post-compromise entries are useless and cannot be used to store any information whatsoever about the activities of $\mathcal{A}$. The scheme we construct should however allow for making the strongest security guarantees possible about the log entries created before t. These guarantees are:

1. $\mathcal{A}$ is unable to learn anything about the contents of any log entry made before t.
2. $\mathcal{A}$ cannot undetectably insert entries into the log at any location corresponding to a point in time before t.
3. $\mathcal{A}$ cannot undetectably modify entries recorded before t.
4. $\mathcal{A}$ cannot undetectably delete entries recorded before t.
5. $\mathcal{A}$ cannot undetectably create a new log file.

As mentioned earlier there is a non-zero probability that the device will detect the intrusion attempt. If this is the case then that means there is a window of time $[t - \epsilon, t]$ in which $\mathcal{U}$ knows it is being compromised but $\mathcal{A}$ has not gained control over it yet. In case this happens, the aforementioned window of time $[t - \epsilon, t]$ should be used to erase keying material and other secrets and leave the log in such a state that $\mathcal{A}$ cannot append new valid entries to the log. What's more, for $\mathcal{T}$ it should be easy to determine from the log's state that illegal operations have occurred. If no intrusion is detected, there is no way to discern malicious entries from valid ones. In that case an external tool or party should look at the contents of each log entry to identify the intrusion after the fact. This is however outside of the scope of this research, the focus of this thesis is solely on maintaining secure audit logs and not on parsing or interpreting their contents.

1.4 RESEARCH QUESTION

Now that we have defined the application setting and desired properties, we can state our research question:

> *How to construct an offline, authenticated, forward secure and confidential logging scheme which is suitable for resource-constrained devices?*

This question can be broken down in multiple parts in order to better clarify the underlying problems:

1. *How to achieve the confidentiality, forward security and authenticity and properties for pre-compromise log entries?*
2. *How to make the scheme suitable for resource constrained devices?*

1.5 CONTRIBUTIONS

Our contributions are as follows:

1. We identify fundamental incompatibilities between existing secure logging solutions and the requirements for our application scenario.
2. We describe multiple commonly occurring real-world scenarios in which nearly all of the existing schemes are either unusable or suffer from severe performance penalties.
3. We present a new secure logging scheme called *Immutable Forward Linked and Sealed logging* (IFLS). Our scheme provides forward-security, authenticity, confidentiality and log stream integrity as well as selective verifiability and disclosure.
4. We present an adaptation of IFLS, *Pseudorandom Indexed Forward Linked logging* (PIFL), which is optimized for flash memory.
5. We evaluate the performance and security of both existing secure logging schemes and our proposed scheme and its adaptations.

1.6 REPORT OUTLINE

This thesis report is structured in the following manner: Firstly, an introduction on secure logging is given, the motivation for this research is provided and the goals we hope to accomplish are stated. In Chapter 2 a background on secure logging schemes and the building blocks hereof are given in the form of several definitions and cryptographic primitives. Chapter 3 describes the current state of secure logging schemes. In Chapter 4 we present our original work, a logging scheme called *Immutable Forward Linked and Sealed logging*. A proof of its security is provided, and its security and performance are compared to similar secure logging protocols. Chapter 5 begins with posing multiple commonly occurring real-world scenarios in which most existing schemes do not function well. Then we present our adaptation of IFLS, *Pseudorandom Indexed Forward Linked logging* (PIFL), which is designed specifically for the aforementioned scenarios. We additionally evaluate how these adaptations affect performance and security. In Chapter 6 we evaluate a software

implementation of IFLS and some of the building blocks of PIFL, to assess whether our schemes are truly fit for resource-constrained devices. Finally, we discuss the findings of this thesis and provide an outlook for future research in Chapter 7.

2

BACKGROUND

This chapter lays the foundation for the later chapters by providing and explaining the necessary cryptographic preliminaries and definitions.

2.1 DEFINITIONS

Definition 1. *Confidentiality* entails protecting information from being disclosed to unauthorized parties. In the context of this thesis, confidentiality means that if there exists a ciphertext $\widehat{M}$ encrypted with a key of length k created at a time T. Then it should be computationally infeasible for any polynomial time bounded adversary to recover the plaintext M at any moment $T < T' < T^k$. Where T^k is a realistic upper bound congruent to k.

Definition 2. *Authenticity and Integrity.* For any log entry created according to the logging scheme presented in this thesis there are (among others) two guarantees we want to make. The first one is **integrity**, which means that the log entry is recorded and maintained in the log file exactly as intended. The second guarantee is that of **authenticity**, any entry in the log is valid if and only if it originated from the logging device itself exactly as it was sent. For most practical applications of cryptography, authenticity nearly always implies integrity [Bel17].

Definition 3. *Forward Security.* A cryptographic scheme is forward secure (or secret) if compromising a long-term key, or a session key at some time in the future does not compromise the security of communications made in the past [MOV96].

Definition 4. *Quality of Forward Security.* Forward security can be implemented in various ways depending on the needs of the user. In the most security sensitive scenario, one would want to update encryption keys after each encryption operation (per-item basis). However, in some cases security is, to some degree, subordinate to performance. In such cases, one might choose to encrypt on an interval basis rather than a per-item basis. This performance-forward security quality trade-off is called "Quality of Forward Security (QoF)" and it is used to indicate the amount of risk a user is willing to accept in return for better performance [Ma08].

Definition 5. *Log Stream Integrity.* In [MT09] Ma *et al.* introduce the concept of *log stream integrity.* Which in addition to the requirements as mentioned in definition 2, adds the additional requirement that log entries cannot be reordered in the log.

2.2 CRYPTOGRAPHIC PRIMITIVES AND MATHEMATICAL TOOLS

2.2.1 *Symmetric-key Algorithms*

A symmetric key algorithm is the composition of two functions, "encrypt" and "decrypt" which use the same cryptographic key to create the ciphertext and plaintext respectively. Formally: $\mathcal{E}_k(M) = \widehat{M}$ and $\mathcal{D}_k(\widehat{M}) = M$, where M is the plaintext and $\widehat{M}$ the ciphertext. There exist two families of symmetric-key algorithms; **stream ciphers** combine the plaintext with a pseudo-random key stream to produce the

ciphertext. **Block ciphers** on the other hand, take a block of bytes at a time and encrypt them as a single unit [Sma15]. In this thesis only block ciphers are used.

2.2.2 *Public Key Cryptography*

A cryptographic system which uses a different key for encrypting than for decrypting is called a public key cryptography scheme (PKC). Usually encryption is done with the public key which may widely distributed, and decryption is done with the private key which is known exclusively to the owner. This construction achieves the confidentiality property (see: definition 1) and the authenticity property (see: Definition 2) [Sma15].

2.2.3 *Signature Schemes*

Digital signatures as described by [GMR88] allows one to sign a message with their private key which can then be verified by any party given the original message and the signer's public key.

Forward Secure Signature Schemes

Forward Secure Signature Schemes are PKC based schemes where the signing key changes over time. For the first time period there exists an initial signing key. Then, at the end of each time period a key evolution function creates the key for the next time period and deletes the old signing key. Signing a message (i.e. signature creation) requires three inputs: the message to be signed, the public key and the current time period. Likewise, for verification of a signature the same three inputs are required and, self-evidently, the signature itself. Formally, a forward-secure digital signature scheme is defined as a quadruple of functions; **Initialize**, **Sign**, **Update**, **Verify**, where:

1. **Initialize**$(k,\ T) \rightarrow (SK_0,\ PK)$ **:** Takes as input the security parameter $k \in \mathbb{N}$ and the number of periods T and returns the initial secret key and public key.
2. **Sign**$(m,\ SK_i) \rightarrow \sigma_i$ **:** Takes the secret key SK_i for the current time period i and the message m and returns the signature σ_i of m for period i.
3. **Update**$(SK_i) \rightarrow SK_{i+1}$ **:** Takes the secret key for the current time period and returns the updated key for the next period.
4. **Verify**$(m,\ \sigma_i,\ PK) \rightarrow \{valid|invalid\}$**:** Takes as input a message m, a putative signature σ_i, the public key PK and returns whether the signature is valid for the given message and time period.

The security guarantee that these signature schemes provide, is that compromisation of the signing key for a given time-period does not enable signature forging for any of the preceding time periods. In terms of forward security this is the absolute strongest guarantee possible [DVW92]. There exist many different Forward Secure Signature schemes, the three most relevant ones will be discussed below.

Bellare & Miner Signature Scheme

Bellare and Miner [BM99] was the first signature scheme addressing forward security for which the space complexity was $O(\log T)$ rather than $O(T)$, where T is the number

of time periods. Bellare and Miner suggest a scheme based on a binary certification tree. In their scheme a binary tree is constructed with T leaves, exactly one leaf for each time period. Each node in the tree corresponds to a key pair instance (PK, SK) of a regular signature scheme. The public key of the Bellare and Miner scheme is the root of the aforementioned tree, the private keys are the set of private signing keys corresponding to all distinct paths from the root to each leaf. A signature for a time period i is constructed by creating a certification chain from leaf i to the root, where the actual message is signed with the private key in the leaf. Each node in the tree is used to sign the public key of its children. Verification of a signature σ_i is done by starting at leaf i and moving upwards through the tree, verifying each signature in the chain up to the root signature, which is verified against the public key.

Malkin Micciancio Miner Signature Scheme

The scheme published by Malkin *et al.* [MMM02] achieves a new feature that previous schemes did not; the number of time periods (T) does not need to be fixed in advance and consequently does not influence performance. Malkin, Micciancio and Miner suggest two composition operations which take any two forward-secure schemes with time periods T_1 and T_2 respectively and construct a new forward-secure scheme with more time periods. These constructions are suggested as tools in constructing flexible forward-secure scheme by applying them repeatedly in different combinations.

The Extended Merkle Signature Scheme

The extended Merkle signature scheme (XMSS) by Buchmann *et al.* [BDH11] is a signature scheme which aside from the forward secure property, also promises to be post-quantum secure. It achieves this property by using a pseudorandom function family and a hash function which is second preimage resistant. With only these very minimal security assumptions (e. g. psuedorandomness and second preimage resistance) it is provably forward secure. The current consensus among the cryptographic community is that the aforementioned properties are not threatened by quantum computers [Ber09] [Fef10] as such the XMSS scheme is post-quantum secure.

XMSS is based on a one-time signature scheme (OTS), which is a signature scheme in which a key pair can only be used once. To create a many-time signature scheme, XMSS uses multiple OTS key pairs and authenticates their public keys trough a Merkle Tree. In this tree the leaves are the hash digests of the OTS public keys and the tree root is the public key. To avoid having to store every OTS key pair individually XMSS uses a pseudorandom function to generate them.

2.2.4 *Cryptographic Hash Functions*

A hash function, formally defined as: $H : \{0,1\}^* \rightarrow \{0,1\}^{|H|}$, takes an arbitrary length input and produces a fixed length alphanumeric output called the "digest" or "hash". A *cryptographic* hash function follows the aforementioned definition and provides some additional guarantees. **(i) Determinism**, the same input always results in the same digest. **(ii) preimage resistance**, it is computationally infeasible to recover the input from a hash. **(iii) second-preimage resistance**, finding two different inputs which produce the same hash is computationally infeasible. **(iv) Non-**

correlation a small change to the input changes the resulting digest so much that the new digest appears uncorrelated with the new one [Sma15].

2.2.5 *Hash Chains*

A hash chain is the result of successively applying a cryptographic hash function to an initial input, commonly referred to as the "chain root"; $x_0 \xrightarrow{H} x_1 \xrightarrow{H} ... \xrightarrow{H} x_n$. As mentioned in Subsection 2.2.4, one of the properties of cryptographic hash function is irreversibility. The implication hereof is that given an arbitrary node x_i in a hash chain, it is infeasible to recover any node $x_j, j < i$.

Hash chains are an integral part of a large number of logging schemes [SK98] [Hol06] [MT09] [CW09] [YNR12] [YN12]. This is due to the fact that hash chains allow for recording chronology, a property which is extremely important to secure logging schemes. An example of how a hash chain can be used to record chronology is the secure time-stamping protocol [HS91]. In this protocol a trusted time-stamping service (TSS) constructs a hash chain by inserting messages into it in the order they arrive. Interested participants can then send and receive messages to the TSS in order to ensure the chronology of these messages.

2.2.6 *Message Authentication Codes*

Message authentication codes (MACs) are short pieces of data appended to messages with which the message's authenticity can be verified. A MAC is created using the message it is appended to and a secret key. The receiver of the message can then verify the received message provided they possess the key used to create the MAC. This key should be shared between the sender and the receiver over a different (secure) channel than the one message has been sent over. The most common MAC implementation is the hashed message authentication code (HMAC) which, as the name implies, uses a cryptographic hash function to create a MAC.

2.2.7 *Aggregate Signatures*

A concept introduced by Ma et. al. [MT09], aggregate signatures allow for authenticating a set of messages with a single data tag. When combined with hash chains they provide a convenient method for achieving forward security and authenticity, while at the same time being computationally- and space efficient. The idea behind aggregate signatures is that the MAC of each individual message is "folded" into a single hash digest. This is done by incrementally hashing the old aggregate signature together with the MAC of the new message to arrive on a new, fixed-length aggregate signature.

Formally, creating an aggregate signature is done as follows: We have an initial secret x_0 as the root of a hash chain and a set of messages $\{m_0, ..., m_n\}$. For the first item we compute $\sigma_{0,0} = H(MAC_{x_0}(m_0))$, then we evolve the secret to the next entry in the hash chain $x_1 = H(x_0)$ and delete x_0. For entries $1 \leq i \leq n$ the following steps are taken:

1. Compute $\sigma_{0,i} = H(\sigma_{0,i-1} \| MAC_{x_i}(m_i))$
2. Evolve x_i to $x_{i+1} = H(x_i)$

3. Delete $\sigma_{0,i-1}$ and α_i

The resulting aggregate signature $\sigma_{0,n}$ is the length of only a single digest and can be used to authenticate the entire set of messages. This is done by sharing α_0 and $\{m_0, ..., m_n\}$ with a receiver (over separate channels) who will then follow the same protocol as described above to verify that the resulting tag is equal to the one received from the sender. It's important to note that aggregate signatures do not require the entire set of messages to be present on creation. As such they are extremely well-suited to work with streams where messages arrive over a period of time. In such a scenario the tag can be incrementally updated on arrival of each new message.

3

STATE OF THE ART

Due to the importance of audit logs in the IT landscape, a myriad of secure logging schemes have been developed throughout the years. All of the different application scenarios for which a logging protocol has been developed would be too broad of a scope to discuss in this chapter. We will therefore limit ourselves to only research directly relevant to this thesis. This will generally consist out of general-purpose logging and resource-constrained logging schemes. In the last section of this chapter (3.6) a comprehensive overview is given of how each of the discussed schemes compares to it counterparts.

3.1 SECURE LOGGING SCHEMES

A secure logging scheme is a cryptographic data structure which is typically defined as the composition of a state and several functions. The entire internal state of a secure logging scheme is represented by $\mathcal{L}_L$ which encompasses the log entries $\{E_1, ..., E_L\}$ but also the current cryptographic keys, hash chain nodes, HMACs or aggregate signatures. Abstracting the internal state away behind a single expression ($\mathcal{L}$) allows for concisely defining the functionality of a secure logging scheme as the arrangement of three functions operating on $\mathcal{L}$:

1. **Initialize** $(S) \rightarrow \mathcal{L}_0$ **:** Takes a set of initialization parameters S and returns the readied internal state.
2. **Append** $(ED,\ \mathcal{L}_{L-1}) \rightarrow \mathcal{L}_L$ **:** Takes some new event's data (ED) and the then current state of the log and creates a new log entry out of ED which is added to the log. Furthermore, cryptographic keys, signatures and so forth are updated or deleted to reflect the new current state. Lastly the function returns the updated state.
3. **Verify**$(s \subseteq S,\ \mathcal{L}_L) \rightarrow \{valid|invalid\}$ **:** Takes the current state of the log and the initialization parameters S (or a subset of S, in case of PKC based schemes) and returns whether the state is valid or not.

There are multiple different classes of secure logging schemes, the most relevant of which will be discussed in the coming sections. Figure 1 gives an overview of these classes.

3.2 SCHNEIER AND KELSEY BASED LOGGING SCHEMES

In 1998 Schneier and Kelsey published "*Cryptographic Support for Secure Logs on Untrusted Machines*" [SK98]. This paper is the foundation upon which the a significant amount of logging schemes in use today are based. [Hol06] [MT09] [YN12] [Wat+04] [AKV03].

In Schneier and Kelsey's scheme an untrusted logger $\mathcal{U}$ creates a new log file with identifier ID and generates a corresponding initial authentication key A_0. $\mathcal{U}$ then irrevocably commits both ID and A_0 to a trusted remote machine $\mathcal{T}$. The reason for

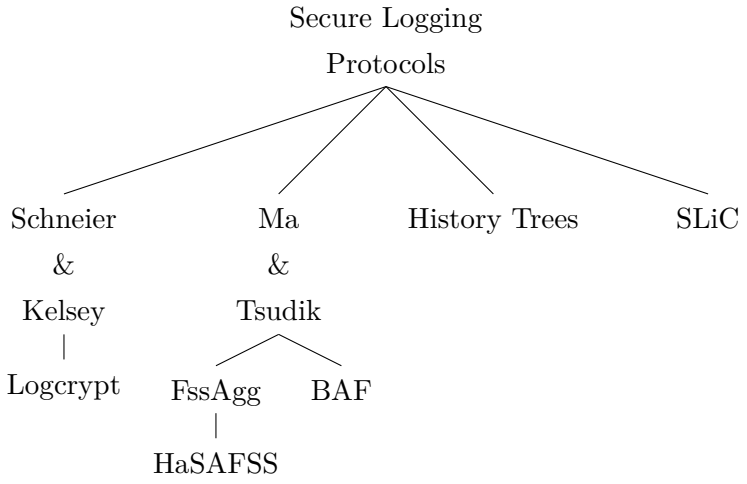


Figure 1: Secure logging scheme classes.

doing this is to prevent an adversary $\mathcal{A}$ from deleting a log file in its entirety only to then claim that no log file was ever created.

After a new log file has been opened and registered with $\mathcal{T}$, $\mathcal{U}$ can add entries to the log as follows: For log entry j with data type $\mathsf{W_j}$

1. Evolve the previous authentication key into the new one: $\mathsf{A_j} = \mathsf{H}(\mathsf{A_{j-1}})$, where $\mathsf{H}(\mathsf{x})$ is some cryptographic hash function.

2. Construct the session key as $\mathsf{K_j} = \mathsf{H}(\mathsf{W_j}, \mathsf{A_j})$.

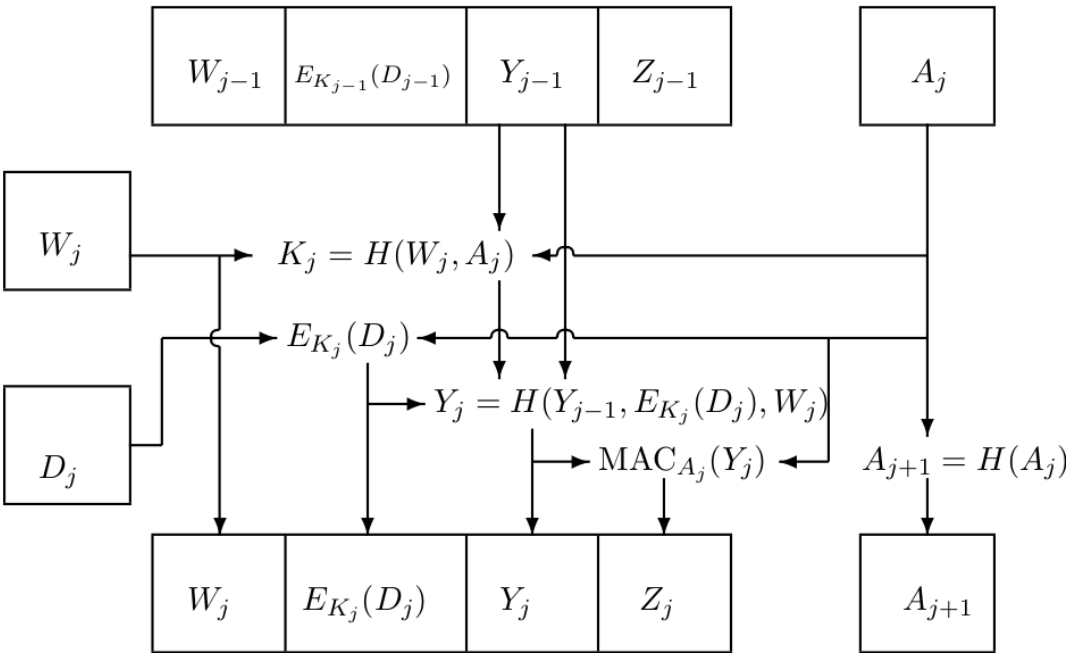


Figure 2: Adding a new entry to the log.

3. Symmetrically encrypt the data under session key $\mathsf{K_j}$.

4. Compute the next entry in the hash chain and its corresponding message authentication code as
$\mathsf{Y_j} = \mathsf{H}(\mathsf{Y_{j-1}}, \mathsf{E_{K_j}}, \mathsf{W_j})$, $\mathsf{Z_j} = \mathsf{MAC_{A_j}}(\mathsf{Y_j})$.

5. Store the newly generated log entry in the following format: $\{W_j, E_{K_j}(D_j), Y_j, Z_j\}$.

To verify a log file, or a subset of its entries, $\mathcal{T}$ establishes a secure channel with a verifier $\mathcal{V}$ and sends A_0 through it. To then verify some log entry $\{W_j, E_{K_j}(D_j), Y_j, Z_j\}$ $\mathcal{U}$ has sent to $\mathcal{V}$, $\mathcal{V}$ performs the following steps:

1. Reconstruct the log entry's authentication key as $A_j = H^{\circ j}(A_0)$.
2. Verify that $Y_j = H(Y_{j-1}, E_{K_j(D_j)}, W_j)$.
3. Verify that $Z_j = MAC_{A_j}(Y_j)$.
4. If either step 2 or 3 fails, abort verification and output the reason.

Despite its popularity, Schneier and Kelsey's scheme does have some drawbacks and vulnerabilities, these will be briefly expanded upon below.

Forging Attack by Verifier

Because $\mathcal{V}$ receives the authentication key from $\mathcal{T}$, a malicious $\mathcal{V}$ could potentially tamper with the entire contents of the log file. The only thing they cannot do is delete the log file outright, because the log's ID is registered with $\mathcal{T}$. However, because they can delete any entry from the log, this makes very little difference in practice.

Delayed Detection Attack

As mentioned earlier, $\mathcal{V}$ cannot verify a log file by itself, it needs to receive A_0 from $\mathcal{T}$. If this is done before $\mathcal{T}$ has received the most recent copy of the current open log on $\mathcal{U}$, and before it has been closed. Adversary $\mathcal{A}$ could tamper with records logged from before they had compromised $\mathcal{U}$ while temporarily avoiding detection. $\mathcal{T}$ will however detect this attack as soon as it has received the updated version of the log file.

Truncation Attack

If $\mathcal{A}$ compromises $\mathcal{U}$ it would be realistic to assume that they have the intention to delete the log entries documenting their break-in. So long as the log file has not been closed, $\mathcal{A}$ could simply delete however many adjoining tail-end records they desire. This is analogous to iteratively deleting the head of the hash chain (Y_n), as such it is impossible for either $\mathcal{V}$ or $\mathcal{T}$ to detect this attack.

Online Server

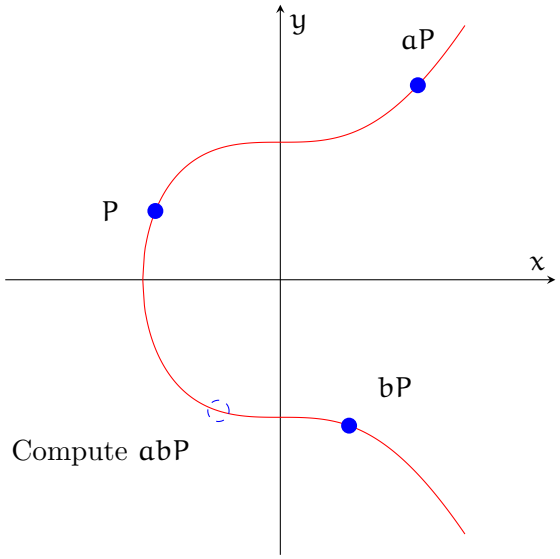
For every new log file opened by $\mathcal{U}$, it has to communicate with $\mathcal{T}$. This is an acceptable scenario for devices for which good connectivity can be assumed, however for a large portion of devices in use even today, this assumption cannot be made. Schneier and Kelsey address this problem as follows: "In essense [sic], this technique is an implementation of an engineering tradeoff between how online $\mathcal{U}$ is and how often we expect $\mathcal{U}$ to be compromised.". However we pose the argument that there are certainly classes of devices where connectivity cannot be assumed, but log file integrity is nevertheless highly desirable. One such example of these devices would be military sensors deployed in hostile environments to gather threat intel for prolonged periods of time.

3.2.1 *Logcrypt*

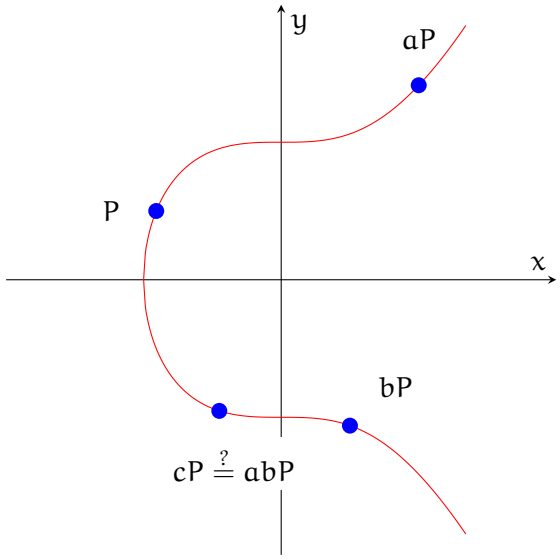
Jason Holt's Logcrypt[Hol06] is one of the secure logging schemes derived from Schneier and Kelsey's scheme. In his paper Holt describes three versions of his scheme, one which uses a pre-shared key between $\mathcal{U}$ and $\mathcal{V}$ or $\mathcal{T}$, one which uses public-key cryptography and one which makes use of Identity-Based Signatures (IBS) [CC03]. For this chapter only the latter will be discussed, because it is the most relevant in the context of this research.

Identity-Based Signatures from Gap Diffie-Hellman Groups

Logcrypt's IBS are constructed from Gap Diffie-Hellman (GDH) groups as described in Cha *et al.* [CC03]. These are groups for which the Decisional Diffie-Hellman problem (Fig. 3b) can be solved in polynomial time, but the Computational Diffie-Hellman problem (Fig. 3a) has no known solution in polynomial time. Tuples of the form (P, aP, bP, cP) which are solutions to the Decisional Diffie-Hellman problem are called valid Diffie-Hellman tuples.



(a) Computational Diffie-Hellman Problem (CDHP): Given (P, aP, bP) find abP.



(b) Decisional Diffie-Hellman Problem (DDHP): Given (P, aP, bP, cP) determine whether $c = ab$.

Figure 3: Illustrations of the two problems underlying Gap Diffie-Hellman Groups.

Let G be a Gap Diffie-Hellman group of prime-order ℓ. Using G, the entire IBS scheme can be concisely defined as the composition of four functions:

1. **Setup:** Let P be a generator of G and let $s \xleftarrow{R} \mathbb{Z}/\ell$ be the master secret. Compute $P_{pub} = sP$ and choose two cryptographic hash functions: $H_1 : \{0,1\}^* \times G \to \mathbb{Z}/\ell$ and $H_2 : \{0,1\}^* \to G$. The parameters of the scheme are then defined as the tuple: $(PKG^{pub} = (P, P_{pub}, H_1, H_2),\ PKG^{priv} = s)$.

2. **Extract:** Given an identity ID, calculate the corresponding private and public keys as: $D_{ID} = sH_2(ID)$ and $Q_{ID} = H_2(ID)$ respectively.

3. **Sign:** Given a private key D_{ID} and some message m, let $r \xleftarrow{R} \mathbb{Z}/\ell$, $U = rQ_{ID}$, $h = H_1(m||U)$ and $V = (r+h)D_{ID}$. The signature of m for identity ID is then given by $\sigma_m^{ID} = (U, V)$.

4. **Verify:** Given a signature $\sigma_m^{ID} = (U, V)$ of a message m for an identity ID. Check whether $(P, P_{pub}, U + hQ_{ID}, V)$ where $h = H_1(m, U)$ is a valid Diffie-Hellman tuple.

Logcrypt uses IBS to attempt to achieve the forward security and authenticity properties, however it does not fully succeed in achieving the former as is demonstrated later on in this section.

The Scheme

In Holt's scheme an untrusted logger $\mathcal{U}$ generates a key generator pair (PKG_0^{pub}, PKG_0^{priv}) and a period length n, which are both irrevocably committed to a trusted computer $\mathcal{T}$. The period length n is the amount of log entries which can be signed with a generator pair before a new pair should be generated. After the first key generator pair is registered with $\mathcal{T}$ the log is maintained according to the following steps:

For message m_i where $i \in \{0, ..., n-1\}$ in period N

1. Let $sk_i = \text{Extract}(PKG_N^{priv}, i)$
2. Let $\sigma_i = \text{Sign}(m, sk_i)$
3. Let $L_i = (m_i, \sigma_i)$
4. Append L_i to the log

For message m_i where $i \bmod n = 0$

1. Generate a new key generator pair $(PKG_{N+1}^{pub}, PKG_{N+1}^{priv})$
2. Let $sk_i = \text{Extract}(PKG_N^{priv}, i)$
3. Let $\sigma_i = \text{Sign}(PKG_{N+1}^{pub}, sk_i)$
4. Delete the old generator pair $(PKG_N^{pub}, PKG_N^{priv})$
5. Let $L_i = (PKG_{N+1}^{pub}, \sigma_i)$
6. Append L_i to the log

Logcrypt achieves the forward security property by sealing a period when another period begins by deleting the private key generator (PKG^{priv}). It's important to note that this is not a particularly strong version of forward security, because an adversary can break in anywhere during the current period and undetectably tamper with all entries in said period. Moreover, there are some other weaknesses present in logcrypt.

Truncation Attack

Like in Schneier & Kelsey's scheme Logcrypt is vulnerable to truncation attacks. Simply by deleting contiguous tail-end messages, an adversary can undetectably delete data from the log. This attack is not limited to the most recent period either, because it is impossible for a verifier to know how many periods were in the log before deletion.

3.3 FORWARD SECURE SEQUENTIAL AGGREGATE SCHEMES

Forward secure sequential aggregate schemes (FssAgg) are schemes based on the authentication technique proposed in [MT07]. In a FssAgg scheme individual log entry signatures are erased once they are folded intro the aggregate signature. Subsequent validity of individual log entries is implied by the validity of the aggregated signature computed over all log entries.

In [MT09] [Ma08] Ma *et al.* introduce four secure logging schemes;

1. **MAC-FssAgg [MT07]:** A private verifiable scheme using aggregate signatures as described in Subsection 2.2.7.
2. **BM-FssAgg [Ma08]:** A publicly verifiable secure logging scheme based based on a modified version of the Bellare-Miner forward secure digital signature scheme [BM99].
3. **AR-FssAgg [Ma08]:** A publicly verifiable secure logging scheme similar to BM-FssAgg, except that it is based on the Abdalla-Reyzin forward secure digital signature scheme [AR00].
4. **iFssAgg [MT09]:** A publicly verifiable scheme similar to MAC-FssAgg with additional verifier efficiency.

The general idea behind each of the above schemes will be expanded upon below. Furthermore, the two classes of schemes derived from FssAgg by Yavuz *et al.*, Hash-Based Sequential Aggregate and Forward Secure Signature (HaSAFSS) [YN12] and Blind Aggregate Forward (BAF) [YNR12], will be discussed as well.

3.3.1 *Bellare-Miner FssAgg & Abdalla-Reyzin FssAgg*

The Bellare-Miner FssAgg (BM-FssAgg) and Abdalla-Reyzin FssAgg schemes use the commutability of the signatures generated by both the Bellare-Miner [BM99] and Abdalla-Reyzin [AR00] forward secure signature scheme to create aggregate signatures. For the exact details on how these signatures are constructed, refer to the paper by Sunitha *et al.* [SA08].

3.3.2 *Message Authentication Code FssAgg*

The MAC-FssAgg is intended for use in scenarios where a logger $\mathcal{U}$ exclusively forwards information to a trusted entity and no interactive communication is needed. The logging scheme is then defined as follows:

1. **Initialize** $(k) \rightarrow sk_0$ **:** Generates a k-bit secret key sk_0.
2. **Sign** $(m_i, sk_i, \sigma_{0,i-1}) \rightarrow \sigma_{0,i}$ **:** Receives the new log item m_i and computes the corresponding MAC as $\sigma_i = MAC_{sk_i}(m_i)$. Then σ_i is folded into the aggregate signature: $\sigma_{0,i} = H(\sigma_{0,i-1} || \sigma_i)$. Subsequently, σ_i is deleted.
3. **Update** $(sk_i) \rightarrow sk_{i+1}$ **:** Takes the current secret key and updates it to the next iteration. The most straight-forward approach for doing this, is with a hash chain (2.2.5).

4. **Verify** $(\sigma_{0,i}, \{m_0, ..., m_i\}, sk_0) \rightarrow \{valid|invalid\}$ **:** Verifies a signature by mimicking the signing process.

The advantages of this scheme are low space- and computational-complexity ($O(1)$ and $O(|H|)$ respectively) [YN12].

3.3.3 *Immutable-FssAgg*

In the MAC-FssAgg scheme validity of individual log entries is implied by the validity of the entire log. This indirect verification is computationally costly if a verifier is only interested in verifying a single entry. Furthermore, if verification were to fail then that tells the verifier only that an entry is wrong somewhere in the log and it says nothing about the veracity of an individual entry.

Immutable-FssAgg attempts to solve the aforementioned shortcomings by keeping individual signatures in the log. Naturally this solution uses more space than MAC-FssAgg, because each log entry is accompanied by a signature.

The iFssAgg scheme can be described as a modification on the MAC-FssAgg scheme:

1. **Initialize** $(k) \rightarrow \{sk_2, \{[(L_0), (L_1, \sigma_1)], \sigma_{0,1}\}\}$**:** Generates a k-bit secret key sk_0 and uses this key to sign a "phantom MAC" on a dummy entry L_0. This MAC is not committed to the log, only L_0 is. Then another dummy event, L_1, is created, on L_1 a MAC is calculated as well. Subsequently the MACs are used to calculate an aggregate signature over L_0 and L_1. Finally (L_1, σ_1) is committed to the log and σ_0 is deleted.

The resulting log file is of the form: $\{[(L_0), (L_1, \sigma_1), ..., (L_i, \sigma_i)], \sigma_{1,i}\}$. By not committing σ_0 an adversary cannot insert a valid entry anywhere in the log, because they need σ_0 to compute a new valid aggregate signature.

3.3.4 *Hash-Based Sequential Aggregate and Forward Secure Signature*

In [YN12] Yavuz *et al.* introduce a class of schemes derived from FssAgg which they call Hash-Based Sequential Aggregate and Forward Secure Signature (HaSAFSS). These schemes are designed to run on Unattended Wireless Sensor Networks (UWSNs), which are low-powered "set it and forget it" devices. At the time of writing the only schemes belonging to the aforementioned class are: **(i)** symmetric HaSAFSS (Sym-HaSAFSS), **(ii)** Elliptic Curve Cryptography (ECC) based HaSAFSS scheme (ECC-HaSAFSS) and **(iii)** self-sustaining HaSAFSS (SU-HaSAFSS) scheme. All of which were created by Yavuz *et al.* themselves.

HaSAFSS schemes allow the signer to compute publicly verifiable, fixed-size and compact signatures efficiently. In the three aforementioned schemes the computational overhead is equal for both verifiers and signers. The way HaSAFSS schemes achieve forward security is by using Timed-Release Encryption (TRE) [CHS07]. The goal of TRE is to encrypt a message in such a manner that no entity can decrypt it until a pre-defined future moment. By using the time factor via timed-release encryption, an asymmetry is introduced between the sender and its receiver(s). This asymmetry allows HaSAFSS schemes to achieve high computational efficiency by

minimizing expensive operations, while still remaining publicly verifiable and forward secure.

The security objective of a HaSAFSS scheme is to achieve secure signature aggregation and *time-valid* forward security simultaneously. This objective varies slightly from that of FssAgg schemes, which aim for *permanent* forward-security. This means that log entries in a HaSAFSS scheme are only existentially unforgeable for a given time period, after which the collected data has to be transferred to a sink. The advantage of this reduced quality of forward security is a reduced computational and storage overhead, as well as public verifiability.

The Sym-HaSAFSS and ECC-HaSAFSS schemes have a linear bound on the number of time periods a signer can create a signature for. As such, all signers need to agree on a fixed data delivery schedule prior to deployment. Su-HaSAFSS on the other hand does not have these limitations; it allows the sender to use an unlimited number of time periods. Additionally each sender can determine their own data delivery schedule without needing to communicate with other signers. The trade-off Su-HaSAFSS makes is that it is computationally- and storage-wise more costly than the other HaSAFSS schemes.

3.3.5 *Blind Aggregate Forward*

The Blind-Aggregate-Forward (BAF) logging scheme proposed in 2009 by Yavuz and Ping [YN09], describes a novel way of making a logging scheme publicly verifiable. To this end BAF uses a trusted third party, which generates an initial private key and a chain of public keys, so that each evolution of the private key corresponds to a link in the chain of public keys. Then, the initial private key is sent to the logger, while the chain of public keys are sent to all verifiers.

The novel idea this scheme presents is a method for generating log entries, which takes very little computing power; creating a log entry requires only a few additions and multiplications in a finite field. Verification on the other hand is far less efficient, it can only be done in an all-or-nothing manner. In other words, individual verification of log entries is not possible, it's only possible to determine whether the log file in its entirety is valid. To address this issue Yavuz and Ping published a new scheme in 2012 called Fast Intermediate Blind Aggregate Forward (FI-BAF) [YNR12]. In FI-BAF, aside from the entire log file signature, each log entry has an additional signature which can be used to verify it. Due to the fact that BAF and FI-BAF do not use standard signature primitives, their solution is much faster than comparable schemes such as Logcrypt (3.2.1) or the FssAgg family (3.3).

3.4 HISTORY TREES

A history tree as described by Crosby *et al.* [CW09] is in essence a versioned Merkle tree [Mer87]. Like in a regular Merkle tree, data is stored in the leaves, each of the internal nodes stores the hash of the subtree below them and the hash of the root covers the entire tree. Unlike a Merkle tree, a history tree allows for new leaves to be appended to the right side of the tree. When such an operation occurs, a new version of the tree is created and the hashes of the internal nodes are recalculated accordingly.

History trees have two features which make them exceedingly useful. Firstly, as with Merkle trees, subtrees containing redundant information can be replaced with a node containing the subtree's hash digest. This allows for efficient space usage without losing the ability to verify data. Secondly, if there exists a version j history tree, it is possible to determine what the root hash would have been as of version $i < j$ by pretending that operations $i + 1$ through j do not exist, and then recomputing the hashes of the internal nodes.

3.5 CRASH ATTACK

The schemes discussed above have all put a lot of effort in theoretical proofs of security, however in practice, log devices have a tendency to crash. Apart from crashes on the operating system level, a logger might have to operate in hostile environments where it can experience power loss, or suffer from overheating. Without making strong assumptions about the underlying architecture of a logger, such as operating system, cache and file system, it is nevertheless plausible that a crash leaves the log file in an inconsistent state. In the context of a secure logging scheme this assertion turns out to be a potential attack vector. In their paper Blass *et al.* describe a new kind of attack on secure logging schemes, which they call the *crash attack* [BN17]. For this attack an attacker detectably tampers with the log file, however they then purposefully crash the logger. For a forensic party inspecting the device afterwards, the resulting data corruption is indistinguishable from the data corruption expected of a device which crashed "normally". As such, it is not possible for the auditing party to definitively determine whether the device has been tampered with.

3.5.1 *Secure Logging with Crash Tolerance*

In their paper Blass *et al.* propose a new protocol called SLiC. The idea of SLiC is to randomize the position of log entries in the logfile using a variation of Donald Knuth's Algorithm P [Knu68]. In doing so an adversary without knowledge of the pseudo-random function used can only tamper with random entries in the log. This means that the probability of an adversary tampering with their intended entry is n^{-1}. Furthermore, it should be possible for a verifier to determine whether any data corruption was a consequence of a regular crash, or a crash attack. With the reasoning being that for a regular crash the latest entries are lost and for the crash attack, random entries are lost.

3.6 OVERVIEW AND COMPARISON OF SECURE LOGGING SCHEMES

Table 1: Overview of various properties of the secure logging schemes discussed in this chapter.

		S & C	(FI-) BAF	Logcrypt	FssAgg		SU-HaSaFSS	SLiC
					iFssAgg	BM & AR		
Computational overhead	Log	$2 \cdot H$	H	$ExpOp + H$			$3 \cdot H$	$2 \cdot H$
	Update	$2 \cdot H$	H	N/A	$ExpOp + H$		H	2·PRF
	Verify	$O(l \cdot H))$	$O(l \cdot (ExpOp + H))$				$O(l \cdot H)$	$O(n \log n)$
	Keygen	$O(1)$	$O(L \cdot (ExpOp + H))$				$O(T \cdot (ExpOp + H))$	2·PRF
Storage overhead	Signer	$O(L \cdot \lvert H \rvert)$	$\lvert K \rvert + \lvert \sigma \rvert$				$O(\lvert sk \rvert + c\lvert H \rvert)$	$O(L \cdot \lvert H \rvert)$
	Verifier	$O(\lvert K \rvert)$	$O(L \cdot \lvert K \rvert)$	$O(K)$			O(S')(pk')	$O(\lvert K \rvert)$
Safe against	Insert	✓	✓	✗	✓	✓	✓	✓
	Delete	✓	✓	✗	✓	✓	✓	✓
	Modify	✓	✓	✗	✓	✓	✓	✓
	Truncate	✗	✓	✗	✓	✓	✓	•
	Forge Att. by $\mathcal{V}$	✗	✗	✓	✓	✓	✓	✗
	Crash Att.	✗	✗	✗	✗	✗	✗	•
Miscellaneous	Online TTP	Yes	Yes	Yes	Yes	Yes	Yes	No

*Table 1 shows the costs associated with processing data items for each of the schemes considered in this chapter. H and PRF denote the cost of doing a single hash operation or pseudo-random function call respectively. $|H|, |K|, |\sigma|, |sk|$ signifies the bit length of a hash digest, symmetric key, signature and a private key respectively. The signing and key update costs are given for a single data item. The cost of key generation is given for the total number of items (i. e. L). Signature verification cost is given for $0 < l < L$ items. The storage cost costs are based on the cryptographic overhead introduced by the schemes, furthermore the assumption is made that the cost of storing the data items is the same for all schemes. The term ExpOp is an abbreviation for *Expensive Operation*, which refers to computations that are expensive to perform, such as modular exponentiation [Sti05] and pairing [Maa04]. The • symbol is used to indicate that a scheme is statistically safe against an attack for large enough values of L. This is a much weaker guarantee than, for instance, "computationally infeasible" provides.

**S & C refers to the scheme by Schneier and Kelsey [SK98]. (FI-) BAF refers to the schemes by Yavuz *et al.* [YN09] [YNR12]. Logcrypt to the scheme by Holt [Hol06]. FssAgg to the Forward Secure Sequential Aggregate class of schemes by Ma *et al.* [MT07]. SU-HaSAFSS to the Sustainable Hash-Based Sequential Aggregate and Forward Secure Signature by Yavuz *et al.* [YN12]. And lastly, SLiC refers to the scheme by Blass & Noubir [BN17].

Table 1 summarizes how each of the schemes discussed in this chapter compares with its counterparts on a variety of principles. From this table the conclusion can be drawn that no scheme satisfies the following criteria: **(i)** secure against all known attacks, **(ii)** does not require an online TTP and **(iii)** computationally lightweight. With this conclusion this chapter has come to an end and a clear goal for the following chapters has been set; designing a secure logging scheme which does satisfy the aforementioned criteria.

4

IMMUTABLE FORWARD LINKED AND SEALED LOGGING

In this chapter the Immutable Forward Linked and Sealed (IFLS) logging scheme is introduced. First the idea behind the scheme will be explained and a formal definition given. Then, we will provide a security proof, after which the scheme's properties are stated and lastly the scheme is compared to other logging schemes.

In the last chapter we had arrived on the conclusion that no existing scheme satisfies the following requirements: **(i)** secure against all known attacks, **(ii)** does not require an online TTP and **(iii)** computationally lightweight. With computationally lightweight we mean that our scheme should be able to run on an embedded platform running other applications besides our logging application. The IFLS scheme presented in this chapter manages to satisfy all of the aforementioned requirements by making use of a novel technique; For each new entry that is appended to the end of the log, an unforgeable reference to the new entry is created in the previous entry. In other words, a forward link gets established. To understand how this is done, we first need to look at the structure of a log entry. When a new log event L_i occurs the corresponding log entry, is a tuple of the form: $(\widehat{L}_i,\ \sigma_i)$. Where $\widehat{L}_i$ is the ciphertext of L_i, symmetrically encrypted under key k_i. And σ_i is the signature of the ciphertext constructed in the following manner: $\sigma_i = \mathcal{H}(\widehat{L}_i \| s_i)$ for some signature key s_i. Then, if another log event L_{i+1} happens, the corresponding log entry is naturally $(\widehat{L}_{i+1},\ \sigma_{i+1})$. However we additionally make a modification to the previous log entry so that it is now of the form $(\widehat{L}_i,\ \sigma^{\dagger}_{i,\ i+1})$. Where $\sigma^{\dagger}_{i,\ i+1}$ is a signature over the ciphertexts of both L_i and L_{i+1}, constructed like so: $\sigma^{\dagger}_{i,\ i+1} = \mathcal{H}(\sigma_i \parallel \widehat{L}_i \parallel s_i)) = \mathcal{H}(\mathcal{H}(\widehat{L}_i \parallel s_i) \parallel \widehat{L}_{i+1} \parallel s_{i+1})$. It's important to note that for the construction of $\sigma^{\dagger}_{i,\ i+1}$ the signature key s_i is not needed, only s_{i+1} is. As such, s_i can be safely deleted immediately after σ_i has been constructed.

In the above description, it is stated that for the construction of each log entry two keys are needed; k_i to encrypt the log event and s_i to seal the previous log entry $(\sigma^{\dagger}_{i-1,\ i})$ and compute the new signature (σ_i). To satisfy the forward security requirement it needs to be computationally infeasible to retrieve k_j or s_j from either k_i or s_i for $i \neq j$. We achieve this property by maintaining a hash chain Y_x in the internal memory of the device (i. e. not in persistent storage) from which the aforementioned keys are derived. To prevent an attacker from deriving keys for past log entries, only the head of the hash chain is stored. So for a logfile of size n, only the node Y_{n+1} will be in memory. An added benefit of using a hash chain in this manner is that a device $\mathcal{U}$ running this scheme will only need to register the chain root Y_1 with a trusted party $\mathcal{T}$ on initialization. After this has been done, no further communication between $\mathcal{U}$ and $\mathcal{T}$ is necessary except when transferring the logfile.

A naive way to derive k_i and s_i would be hashing Y_i along with two different string constants for each of the keys. The problem with this approach lies with the fact that on some platforms hashing can be a quite expensive operation to perform [NM02]. For this reason it might be a better option to, for example, use a single SHA512 digest and use 128 bits for k_i, 128 bits for s_i and the other 256 bits for computing

the next node in the hash chain. By using a hash chain to derive cryptographic keying material for each new log entry a big step is taken towards satisfying the log stream integrity property. Due to the fact that deletion or insertion of malicious log entries would mean that the logfile is "out-of sync" with the hash chains in terms of keying material. This would get noticed immediately upon verification. Furthermore undetectably tampering with the ith log entry would require either obtaining both k_i and s_i or finding multiple collisions in the used hash function. In Section 4.4 we prove this formally.

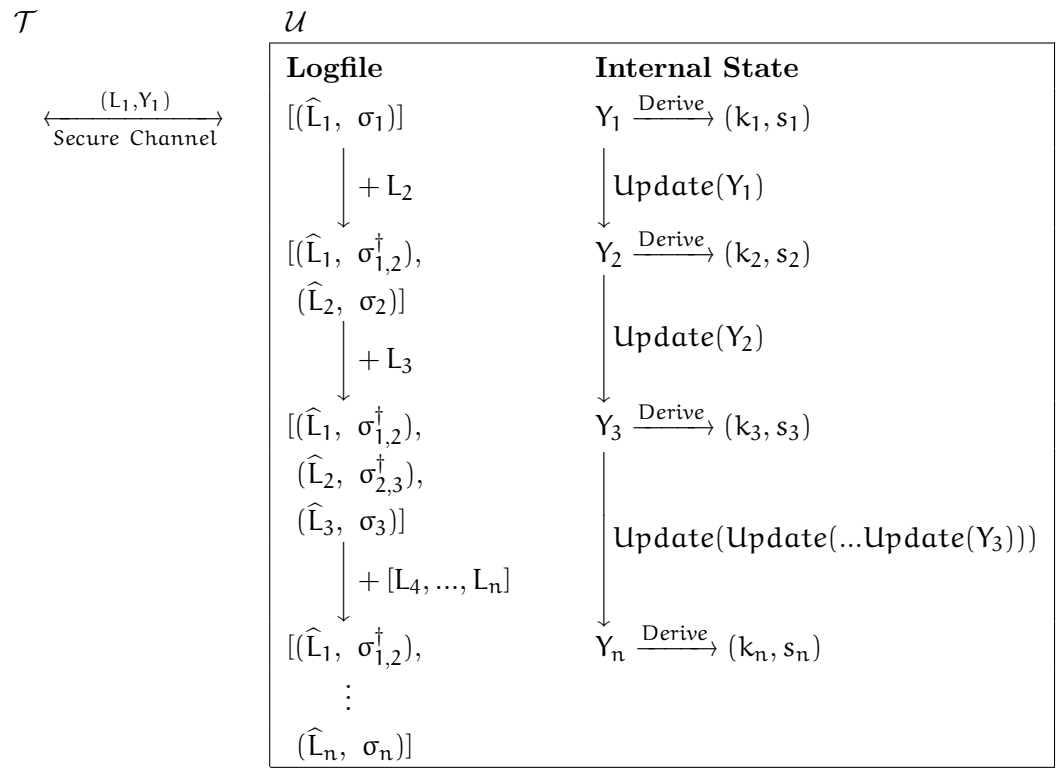


Figure 4: IFLS Scheme

To complete the log stream integrity requirement, we also need to make sure that truncating the logfile is not possible for an adversary. To understand how IFLS prevents this, one must look at the structure of a valid logfile constructed according to the technique described in the previous paragraph:

$$\begin{aligned}\mathcal{L}_N = [&(\widehat{L}_1,\ \sigma^{\dagger}_{1,2}),\\ &(\widehat{L}_2,\ \sigma^{\dagger}_{2,3}),\\ &\quad\vdots\\ &(\widehat{L}_{n-1},\ \sigma^{\dagger}_{n-1,\,n}),\\ &(\widehat{L}_n,\ \sigma_n)]\end{aligned}$$

We start by noting that a log file is valid if and only if it is of the above form. Aside from some specific corner cases, as we explain in Subsection 4.4.5, verification of a logfile of any other form will not succeed. To understand why this prevents truncation

attacks, we look at what happens when an attacker deletes the last k log entries. The resulting logfile would be as follows:

$$\begin{aligned}\mathcal{L}'_{N-k} = [&(\widehat{L}_1,\ \sigma^{\dagger}_{1,2}),\\ &(\widehat{L}_2,\ \sigma^{\dagger}_{2,3}),\\ &\quad\vdots\\ &(\widehat{L}_{n-k-1},\ \sigma^{\dagger}_{n-k-2,\ n-k}),\\ &(\widehat{L}_{n-k},\ \sigma^{\dagger}_{n-k-1,\ n-k})]\end{aligned}$$

Verification of the last log entry would not succeed in this case, because it is equal to $(\widehat{L}_{n-k},\ \sigma^{\dagger}_{n-k-1,\ n-k})$ and not to $(\widehat{L}_{n-k},\ \sigma_{n-k})$. The function of the last log entry is thusly to act as a "seal" of sorts which prevents the log from being undetectably truncated. By negating the possibility of truncation attacks, we have achieved the log stream integrity property.

In Section 1.3 we mentioned that $\mathcal{U}$ might have some form of Intrusion Detection System (IDS). If $\mathcal{U}$ is taken over by an attacker at a time t, and the IDS has detected this compromise, then there is a window of time $[t-\epsilon,\ t]$ in which $\mathcal{U}$ knows it is being compromised but $\mathcal{A}$ has not gained control over it yet. Without making assumptions about the size of this window, we describe a set of steps which $\mathcal{U}$ needs to take to make it infeasible for the attacker to keep logging and, if possible, record the intrusion in the log. This protocol can be interrupted at any point after step one, and the attacker would not gain an attack advantage. The solution we employ, is that we keep a special intrusion signature in memory which is created during the initialization phase of the protocol. This signature is created as follows: $\sigma_! = \mathcal{H}(\text{“intrusion”} \parallel s_!)$, where: $s_! \xleftarrow{\text{Derive}} Y_1$. $s_!$ is discarded as soon the intrusion signature has been created.

If during the logging phase an intrusion is detected, first all the keying material currently in memory is deleted, then the protocol attempts to update the signature of the last log entry with the intrusion signature: $\sigma^{\dagger}_{n,\ !} = \mathcal{H}(\sigma_n \parallel \sigma_!)$. The resulting log file would then be of the following form:

$$\begin{aligned}\mathcal{L}_N = [&(\widehat{L}_1,\ \sigma^{\dagger}_{1,2}),\\ &(\widehat{L}_2,\ \sigma^{\dagger}_{2,3}),\\ &\quad\vdots\\ &(\widehat{L}_{n-1},\ \sigma^{\dagger}_{n-1,\ n}),\\ &(\widehat{L}_n,\ \sigma^{\dagger}_{n,\ !})]\end{aligned}$$

As mentioned earlier, this is not a valid log file and as such this would be flagged by the verifier. The verifier then checks whether the last signature equals the intrusion signature, after which it can conclude that an intrusion has occurred. The exact protocol for an intrusion is the following set of steps:

1. Delete the current signature- and encryption keys, and the hash chain node: $\mathtt{delete}(Y_{n+2},\ s_{n+1},\ k_{n+1})$.
2. Compute the intrusion signature over the last log entry: $\sigma^{\dagger}_{n,\ !} = \mathcal{H}(\sigma_n \parallel \sigma_!)$.
3. Delete σ_n.

4. Write $\sigma^{\dagger}_{n,\ !}$.

If $\mathcal{A}$ interrupts the above protocol before step 1, then they have successfully circumvented the IDS. In the case that $\mathcal{A}$ manages to stop the protocol after Step 1 or 2, then the log file is in a valid state, however $\mathcal{A}$ cannot continue logging, because the hash chain has been deleted. An interruption after step 3 would mean that the log is in an invalid state, which $\mathcal{A}$ cannot revert, because they lack the necessary keys. This scenario is functionally equivalent to $\mathcal{A}$ performing a Crash Attack (4.4.5).

4.1 VERIFICATION AND DISCLOSURE

One of the goals when designing this scheme was separation of disclosure (reading the log's contents) and verification. The reason we pursued this goal can best be explained with an example: A manufacturer builds a logging device $\mathcal{U}$ for an external party, such as a branch of the military. If $\mathcal{U}$ malfunctions, it needs to be sent back to the manufacturer so that they can review $\mathcal{U}$'s activities and diagnose any problems. In this scenario, if verification and disclosure were coupled, then the manufacturer would be able to tamper with the log, due to the fact they have the ability to create valid log entries. In other words *forging attack by verifier* is possible in this scenario.

In the IFLS scheme this scenario is counteracted by having each log entry have two keys associated with it; the encryption key k and the signature key s. This design allows the trusted party $\mathcal{T}$ to send a subset of the logfile $\mathcal{L}_{i,j}$ to either an inspecting party $\mathcal{I}$ or a verifying party $\mathcal{V}$. These parties would additionally receive respectively a subset of the encryption keys $K = [k_i, ..., k_j]$ or a subset of the signature keys $S = [s_i, ..., s_j]$. In doing so, $\mathcal{I}$ is only able to decrypt the contents of the log and similarly $\mathcal{V}$ is only able to verify the log. Effectively preventing the possibility of a forging attack by either $\mathcal{I}$ or $\mathcal{V}$.

4.2 IFLS FORMAL DEFINITION

IFLS consists out of the composition of five functions (initialize, log, update, verify and disclose) and a state (Γ_N). The state encompasses both the logfile $\mathcal{L}_N$ and the internal state Y_N so that $\Gamma_N = \{\mathcal{L}_N, Y_N\}$.

1. **Initialize** $(L_1,\ Y_1) \rightarrow \Gamma_1$ **:** Takes an initialization entry (L_1) and a hash chain root (Y_1) and creates the first log entry ($\widehat{L_1} = \mathcal{E}_{k_1}(L_1), \sigma_1 = \mathcal{H}(\widehat{L_1} \| s_1)$). The encryption key k_1 and the signature key s_1 are securely derived from the hash chain root in a manner most suited to the device IFLS is used on. After $\widehat{L_1}$ has been created, L_1 is deleted from memory. Lastly the `update` function is called on Y_1 and the initialized log state is returned so that $\Gamma_1 = \{\mathcal{L}_1 = [(\widehat{L_1}, \sigma_1)],\ Y_2\}$.

2. **Update** $(Y_i) \rightarrow Y_{i+1}$ **:** Evolves the current hash chain node to its next iteration and returns it.

3. **Log** $(L_i,\ \Gamma_{i-1}) \rightarrow \Gamma_i$ **:** Takes a new log message L_i and creates a new log entry ($\widehat{L_i} = \mathcal{E}_{k_i}(L_i),\ \sigma_i = \mathcal{H}(\widehat{L_i} \| s_i)$). Then the ciphertext of the new log message ($\widehat{L_i}$) is folded into the signature of the previous log entry in order to seal it and create the forward link; $\sigma^{\dagger}_{i-1,i} = \mathcal{H}(\sigma_{i-1} \parallel \widehat{L_i} \parallel s_i)$. Subsequently L_i and σ_{i-1} are deleted and the update function is called to evolve the current hash chain node. Lastly the new log state is returned, which is of the form: $\mathcal{L}_i = \{[(L_0, \sigma^{\dagger}_{0,1}), (L_1, \sigma^{\dagger}_{1,2}), ..., (L_i, \sigma_i)],\ Y_{i+1}\}$.

4. **Verify** $(\mathcal{L}_{i,j},\ [s_i, ..., s_j]) \rightarrow \{valid|invalid\}$ **:** Takes a subset of the logfile and a set of signature keys corresponding to that subset and mimics the logging process. Verifying the signatures at each step to check whether the log state is valid.

5. **Disclose** $(\mathcal{L}_{i,j},\ [k_i, ..., k_j]) \rightarrow [L_i, ..., L_j]$: Takes a subset of the logfile and a set of encryption keys corresponding to that subset and returns the decrypted log entries.

4.3 KEY DERIVATION

The strength of a cryptographic key is commonly expressed in terms of *number of bits of entropy*. A single bit of entropy can be seen as the outcome of fair coin toss [Sha01]. Extending this concept, an n-bit full-entropy key means each bit of the key is chosen independently of the others by the equivalent of a fair coin toss.

The security of a log entry is only as strong as the keys used to create it are. As mentioned earlier in this chapter, a trade-off can be made between the strength of the keys and the performance of the scheme. Or more precisely, between the entropy of the keys and the number of hash operations needed to produce these keys. We present three functions for key derivation; **(i)** *hi-entropy-kd* **(ii)** *med-entropy-kd* and **(iii)** *lo-entropy-kd*. Each of these functions takes as input the current hash chain node Y_i and outputs a tuple of three elements (s_i, k_i, Y_{i+1}) containing the computed signature- and encryption keys, and the next node in the hash chain. The *hi-entropy-kd* function yields very strong keys at expense of performance, *med-entropy-kd* produces medium strong keys for improved performance and *lo-entropy-kd* generates comparatively weak keys in exchange for using only a single hash operation.

1. ***hi-entropy-kd :*** This function produces the aforementioned tuple in the following manner:

$$(s_i \xleftarrow[C_s]{\mathcal{H}} Y_i,\ k_i \xleftarrow[C_k]{\mathcal{H}} Y_i,\ Y_{i+1} \xleftarrow{\mathcal{H}} Y_i)$$

   We create both s_i and k_i by hashing the current hash chain node together with a string literal unique for the type of key being produced, C_s and C_k respectively. We do this to prevent s_i, k_i and Y_{i+1} all being equal to one another. All of the elements of the tuple produced by *hi-entropy-kd* will have information entropy equal to Y_i. In practice this means that, given Assumption 2, s_i, k_i and Y_{i+1} each have exactly $|\mathcal{H}|$ bits of entropy.

2. ***med-entropy-kd* :** This function hashes the current hash chain node with a string literal C_{sk} to produce an intermediary digest which is then split to produce both the signature and the encryption key. This means that both keys have $\frac{1}{2}|\mathcal{H}|$ bits of entropy. *med-entropy-kd* performs one hash operation to produce the keys and another to update the hash chain node.

$$sk_i \xleftarrow[C_sk]{\mathcal{H}} Y_i$$
$$(s_i \xleftarrow{\text{split}} sk_i,\ k_i \xleftarrow{\text{split}} sk_i,\ Y_{i+1} \xleftarrow{\mathcal{H}} Y_i)$$

3. ***lo-entropy-kd* :** In this function we use the current chain node and split it into three pieces so that 1/4th of the digest is used for s_i, 1/4th for k_i and 2/4th

of the digest, y_i, is used only when evolving the node to its next iteration, so that $Y_{i+1} = \mathcal{H}(y_i \parallel s_i \parallel k_i)$. By keeping one half of the hash chain node unused for key derivation we ensure that even if s_i and k_i are compromised, predicting future keys is still computationally hard. Formally *lo-entropy-kd* is expressed in the following manner:

$$Y_{i-1} \xrightarrow{\mathcal{H}} Y_i = (y_i,\ s_i,\ k_i)$$

4.4 SECURITY ANALYSIS

In this Section we prove the security of the IFLS scheme. Our approach for doing this is to first prove the security of the various building blocks used in IFLS. Then, we go over each of the attacks relevant to our scheme, as listed in Table 1, and prove why they are infeasible.

There are four different kinds of parties in our scheme:

1. $\mathcal{U}$ is an untrusted logger which is not sufficiently tamper-resistant or physically secure to prevent an attacker from taking it over. It does not behave maliciously unless it is under the control of an attacker. $\mathcal{U}$ only interacts with $\mathcal{T}$ during its initialization phase, or when it is queried by $\mathcal{T}$.

2. $\mathcal{T}$ is a trusted computer in a secure location. It has the ability to authorize a verifier $\mathcal{V}$ or an Interpreter $\mathcal{I}$ to access the entire, or a subset of the audit log $\mathcal{L}_{i,j}$. It additionally supplies $\mathcal{V}$ with the set of signature keys $S = [s_i, ..., s_j]$ corresponding to the part of the log accessed. Similarly $\mathcal{I}$ receives the set of encryption keys $[k_i, ..., k_j]$.

3. $\mathcal{V}$ is a semi-trusted verifier which verifies the logfile on $\mathcal{U}$. $\mathcal{V}$ can obtain a copy of the logfile and the corresponding signature keys from $\mathcal{T}$. It can however not read any data from the log.

4. $\mathcal{I}$ is a semi-trusted Interpreter of the log on $\mathcal{U}$. Much like $\mathcal{V}$, $\mathcal{I}$ can obtain a copy of the logfile and the corresponding encryption keys from $\mathcal{T}$. It cannot verify any of the log entries.

The adversarial model which is used for IFLS is similar to the *covert adversaries* model by Aumann *et al.* [AL10]. In this model an adversary $\mathcal{A}$ is assumed who is fully malicious, but wants to undetectably achieve a goal. An example of such an adversary in the case of LANDroids, would be an insurgent who has captured one and wants to mislead the military by adding false information to the log to conceal their activities.

Definition 6. The security of the IFLS scheme is defined as the non-existence of a polynomial time bounded adversary $\mathcal{A}$ who compromises $\mathcal{U}$ at a time T and produces an existential forgery of a subset of the logfile $\mathcal{L}_{i,j}$. Where $1 \leq i < j$, and $\mathcal{L}_{i,j}$ corresponds to the part of the audit log created before T.

4.4.1 *Building Blocks*

Before we can prove that our scheme is safe against known attacks, we first prove that the building blocks our scheme uses are secure. To this end we start with proving the security of the hash chain. Then we show how to safely and efficiently derive keys

from a hash chain, and lastly we show that both the ciphertext and the signature of a log entry are secure.

The backbone of our scheme is the hash chain which is maintained in the internal memory of $\mathcal{U}$. Because keys are derived from each node in the chain, it is important that a cryptographic hash function is chosen which is **(i)** preimage resistant, **(ii)** second preimage resistant, **(iii)** non-correlated and **(iv)** deterministic (see also 2.2.4). It would be tempting to model the hash chain in the Random Oracle Model, however because of the issues with this model [CGH04] we choose not to. Instead we will show that obtaining Y_i from Y_j for $j < i$, or finding a value $X' \neq Y_{i-1}$ so that $\mathcal{H}(X') = Y_i$ are both computationally infeasible.

Assumption 1. *The cryptographic primitives used in IFLS for hashing, encryption and MACs have semantic security properties [Gol09] as follows:*

(i) *$\mathcal{H}$ is a secure hash function with properties: (a) determinism, (b) preimage resistance, (c) second-preimage resistance and (d) non-correlation.*

(ii) *$\mathcal{H}(m \parallel s)$ is equivalent to the message authentication code of m with key s and is Existential Unforgeable Under Chosen Message Attacks (EU-CMA).*

(iii) *$\mathcal{E}$ is a symmetric encryption function which is Indistinguishable under Chosen Ciphertext Attacks (IND-CCA secure)*

Assumption 2. *The root of the hash chain (Y_1) is a bit-string with at least $|\mathcal{H}|$ bits of entropy.*

Lemma 1. *Using the algorithm from Assumption 1 to hash at least $|\mathcal{H}|$ bits of entropy, produces a digest with exactly $|\mathcal{H}|$ bits of entropy.*

Proof. By applying Assumptions 1 and 2 and Lemma 1, it can be deduced that any hash chain node Y_i will have exactly $|\mathcal{H}|$ bits of entropy. This means that given Y_i, finding Y_j for $j < i,\ i, j \in \mathbb{N}$, so that $\mathcal{H}^{i-j}(Y_j) = Y_i$ has exponential time complexity $O(2^{|\mathcal{H}|})$. Which is computationally infeasible for all values of i and j, and conventional hash digest sizes [Bel06].

Finding a collision in the hash function so that $\mathcal{H}(X') = Y_i$ for $X' \neq Y_{i-1}$ amounts to performing a birthday attack on the hash function [BK04]. A birthday attack has time-complexity $O(\sqrt{2^{|\mathcal{H}|}})$ which is likewise computationally infeasible for conventional digest sizes. □

The next building block we need to prove secure are the cryptographic keys used to encrypt and sign log entries with. Each of the three key derivation functions described in Section 4.3 satisfy the forward security requirement which dictates that compromising log entry keys s_i or k_i does not compromise the security of past or future log entries. For the proof presented below we will assume that an adversary is in possession of s_i and k_i and wants to obtain a valid key tuple $(s_{i\pm j},\ k_{i\pm j}),\ j > 0$.

Proof. For *hi-entropy-kd* and *lo-entropy-kd*, finding a valid key tuple $(s_{i\pm j},\ k_{i\pm j}),\ j > 0$ is essentially the same problem as finding the exact input for a hash operation given only the digest. In other words, an adversary would need to find Y_i given only $\mathcal{H}(Y_i \parallel S)$ where S is some known string literal. It's important to note that merely finding a collision in this scenario is not sufficient, because a collision would not produce correct keys nor can it be used to compute future hash chain nodes. For this reason we conclude that even in the simplest case $j = 1$, finding a valid key tuple

from another tuple requires reversing at least one hash digest which has computational complexity $O(2^{|H|})$.

In the case of the *lo-entropy-kd* function, an adversary is already in possession of half of the hash input $(s_i,\ k_i)$ and needs only to find the other half. This effectively halves the search space which means the computational complexity becomes $O(\sqrt{2^{|\mathcal{H}|}})$. For conventional hash digest sizes both of the aforementioned bounds are considered secure. □

A log entry consists out of two building blocks, the ciphertext $\widehat{L}_i$ and either a regular signature σ_i, or a forward linked one $\sigma^{\dagger}_{i,i+1}$. Both of these signatures can be regarded as HMACs as described in [KCB97]. HMACs have been proven secure given a secure hashing algorithm (Assumption 1) and a secure key [Bel06], which we have proven to be true earlier. Therefore we can now conclude that signatures used in IFLS are secure. Similarly, if we assume a secure key and apply Assumption 1, we can conclude that the ciphertext is secure as well.

Now that we have proven that each of the building blocks of IFLS are secure, the next step is proving that our scheme is safe against all the attacks listed in Table 1.

4.4.2 *Insertion and Deletion attacks*

Undetectably inserting or deleting a log entry in the pre-compromise part of the logfile is as hard as reversing at least one hash chain node (Proof 4.4.1). For this scenario we will assume an adversary $\mathcal{A}$ who wants to delete or insert one or more log entries into $\mathcal{L}_{1,j}$ where j corresponds to the last entry of the pre-compromise logfile. It is important to note that this scenario does not consider deleting the last n messages of the log file, because that is different kind of attack called the "truncation attack" (4.4.3).

Proof. In the simplest case, $\mathcal{A}$ deletes a single log entry $(\widehat{L}_i, \sigma^{\dagger}_{i,i+1})$ from the logfile. Said logfile is now of the following form:

$$\begin{aligned}\mathcal{L}'_N = [\quad & \vdots \\ & (\widehat{L}_{i-1},\ \sigma^{\dagger}_{i-1,i}), \\ & (\widehat{L}_{i+1},\ \sigma^{\dagger}_{i+1,\ i+2}), \\ & \vdots \qquad\qquad]\end{aligned}$$

Upon verification of $(\widehat{L}_{i-1}, \sigma_{i-1,i})$, $\mathcal{V}$ would check **(i)** whether $\sigma_{i-1,i} \stackrel{?}{=} \mathcal{H}(\mathcal{H}(\widehat{L}_{i-1} \parallel s_{i-1}) \parallel \widehat{L}_{i+1} \parallel s_i)$ and **(ii)** whether $\sigma_{i,i+1} \stackrel{?}{=} \mathcal{H}(\mathcal{H}(\widehat{L}_{i+1} \parallel s_i) \parallel \widehat{L}_{i+2} \parallel s_{i+1})$. Both of these checks would fail because $\widehat{L}_i \neq \widehat{L}_{i+1}$ and $s_i \neq s_{i+1}$. For deletion of multiple log entries, these checks fail as well.

If log entries are inserted, the reasoning is very similar to the above. The logfile now looks as follows:

$$\begin{aligned}\mathcal{L}'_N = [\quad & \vdots \\ & (\widehat{L}_i,\ \sigma^{\dagger}_{i,i+1}), \\ & (\widehat{L}_i,\ \sigma^{\dagger}_{i,i+1})', \\ & (\widehat{L}_{i+1},\ \sigma^{\dagger}_{i+1,\,i+2}), \\ & \quad \vdots \qquad\qquad]\end{aligned}$$

Where $(\widehat{L}_i,\ \sigma^{\dagger}_{i,i+1})'$ is the inserted log entry. In that case verification of all log entries $(\widehat{L}_j,\ \sigma^{\dagger}_{j,j+1})$ for $j > i$ would fail because the signature key used for verification is not the same as the one used to create the entries. This is due to the fact that the log file is not synchronized to the hash chain anymore. □

4.4.3 *Truncation Attack*

Due to the nature of a hash chain, it is impossible to know how often a chain has been evolved if no evidence of the nodes in the chain gets recorded. In the IFLS scheme we record evidence of a node in the form of the ciphertext and signatures created with keys derived from said node. This approach works very well against deletion and insertion attacks, as we have demonstrated in the previous Section. However the problem with the truncation attack is, that by deleting a contiguous end of tail-end log entries, there is now way to know how many log entries were in the log originally by just looking at the hash chain. To address this problem we use two kinds of signatures in IFLS, $\sigma^{\dagger}_{i,i+1}$ for all log entries $i \neq n$, and σ_n for the last entry in the log file. The motivation for doing this is that if an attacker were to delete the last log entry, they would need to construct a new last signature σ'_n. Doing so is computationally intractable, as is shown in the following proof. For this proof we assume an attacker $\mathcal{A}$ who has compromised $\mathcal{U}$ and is now in possession of Y_{n+1}, $\mathcal{A}$'s objective is to delete the last k log entries undetectably.

Proof. An attacker who deletes k entries from the log, leaves the log in the following state:

$$\begin{aligned}\mathcal{L}'_{N-k} = & [(\widehat{L}_1,\ \sigma^{\dagger}_{1,2}), \\ & (\widehat{L}_2,\ \sigma^{\dagger}_{2,3}), \\ & \quad \vdots \\ & (\widehat{L}_{n-k-1},\ \sigma^{\dagger}_{n-k-2,\,n-k}), \\ & (\widehat{L}_{n-k},\ \sigma^{\dagger}_{n-k-1,\,n-k})]\end{aligned}$$

Upon verification of $\mathcal{L}'_{N-k}$, $\mathcal{V}$ would notice that $\sigma_{n-k-1,\,n-k} \neq \mathcal{H}(\widehat{L}_{n-k} \parallel s_{n-k})$. For $\mathcal{A}$ to be able to truncate undetectably, they would need to forge a new last signature σ'_{n-k}. Without having knowledge of s_{n-k}, this entails to finding a collision in the hash function so that $\sigma'_{n-k} = (\widehat{L}_{n-k} \parallel s_{n-k})$. This has computational complexity $O(\sqrt{2^{|\mathcal{H}|}})$. □

4.4.4 *Forging Attack by Verifier or Interpreter*

For this attack we will be assuming an attacker $\mathcal{A}$ masquerading as either $\mathcal{V}$ or $\mathcal{I}$, however not as both. If $\mathcal{V}$ and $\mathcal{I}$ do collude with one another IFLS is not safe anymore and existential forgeries against the log could be made. Furthermore, we assume that if $\mathcal{V}$ notices that a part of the log is not verifiable, then they report it to $\mathcal{T}$. Similarly if $\mathcal{I}$ cannot decrypt a ciphertext, they notify $\mathcal{T}$ as well. In the IFLS scheme it is computationally infeasible for an attacker posing as either a Verifier of Interpreter to create an existential forgery of a subset of the log file.

Proof. An attacker who wants to forge a log entry $(\widehat{L}_i, \sigma^{\dagger}_{i,i+1})$ and is in possession of $[s_{i-1}, s_i, s_{i+1}]$ has everything they would need to create a valid signature of $\widehat{L}_i$. However, they are not in possession of the encryption key k_i and have no computationally feasible way to obtain k_i. For this reason it is not possible for $\mathcal{V}$ to forge a complete log entry $(\widehat{L}_i, \sigma^{\dagger}_{i,i+1})$.

Equivalently, a malicious Interpreter $\mathcal{I}$ who wants to forge a log entry $(\widehat{L}_i, \sigma^{\dagger}_{i,i+1})$ and is in possession of k_i has the ability to create a valid ciphertext $\widehat{L}_i$. However, because they are not in possession of s_i, s_{i-1} and s_{i+1} they cannot create valid signatures and thus $\mathcal{I}$ cannot forge a complete log entry $(\widehat{L}_i, \sigma^{\dagger}_{i,i+1})$. □

4.4.5 *Crash Attack*

The crash attack [BN17] is an attack where the adversary removes log entries from the log file and then crashes the logger. The state of the log after a crash attack is then indistinguishable from the state after a real crash and as such the adversary is capable of undetectably removing entries.

In the IFLS scheme there are three kinds of operations which interact with data stored in persistent storage. These operations are:

1. **Write** $(\widehat{L}_n,\ \sigma_n)$
2. **Delete** σ_{n-1}
3. **Write** $\sigma^{\dagger}_{n-1,n}$

If $\mathcal{U}$ were to crash during one of these operations, it is to be expected that the associated data is lost. Without making any strong assumptions about the underlying storage system $\mathcal{U}$ uses to implement these operations, we make the following assumptions:

Assumption 3. *If data is not involved in a storage operation, such as writing or deleting, then we do not expect this data to be lost in the event that $\mathcal{U}$ crashes.*

Assumption 4. *Writing new log entries, as well as updating signatures are atomic operations up until the moment $\mathcal{A}$ compromises $\mathcal{U}$. In other words, when $\mathcal{A}$ gains access to the internals of $\mathcal{U}$, it will always find the log file in the valid state $\mathcal{L}_N$.*

The above Assumptions leads us to the following proposition:

Lemma 2. *By applying Assumption 3 and 4 we conclude that from the moment $\mathcal{A}$ has gained over $\mathcal{U}$ log entry elements which are affected by file system operations are $\widehat{L}_{n+1},\ \sigma_{n+1},\ \sigma^{\dagger}_{n,n+1}$.*

To prove that our scheme is secure against the Crash Attack we assume an adversary $\mathcal{A}$ who compromises $\mathcal{U}$, deletes one or more items from the log and then crashes $\mathcal{U}$ to attempt to hide their activities. $\mathcal{A}$ is considered successful if $\mathcal{V}$ cannot determine whether the state of the log file is the result of a regular crash, or from a crash attack.

Proof. Using Assumptions 3, 4 and Lemma 2 we deduce that a regular crash can leave the log file in one of the following states:
State 1

$$\begin{aligned}\mathcal{L}_N^{Cr} = [& \quad \vdots \\ & (\widehat{L}_n,\ \sigma^{\dagger}_{n,\ n+1}), \\ & (\widehat{L}_{n+1},\ \varnothing)]\end{aligned}$$

State 2

$$\begin{aligned}\mathcal{L}_N^{Cr} = [& \quad \vdots \\ & (\widehat{L}_n,\ \sigma^{\dagger}_{n,\ n+1}), \\ & (\varnothing,\ \varnothing)]\end{aligned}$$

State 3

$$\begin{aligned}\mathcal{L}_N^{Cr} = [& \quad \vdots \\ & (\widehat{L}_n,\ \varnothing), \\ & (\widehat{L}_{n+1},\ \sigma_{n+1})]\end{aligned}$$

State 4

$$\begin{aligned}\mathcal{L}_N^{Cr} = [& \quad \vdots \\ & (\widehat{L}_n,\ \varnothing), \\ & (\widehat{L}_{n+1},\ \varnothing)]\end{aligned}$$

State 5

$$\begin{aligned}\mathcal{L}_N^{Cr} = [& \quad \vdots \\ & (\widehat{L}_n,\ \varnothing), \\ & (\varnothing,\ \varnothing)]\end{aligned}$$

If upon verification $\mathcal{L}_N$ is in any other state than a valid state, or one of the five states listed above, then it can immediately be considered as tampered with. In this case $\mathcal{A}$ has failed in hiding its attack.

It is important to note at this point that $\widehat{L}_n$ is the last ciphertext of the pre-compromise part of the log file. Therefore we are now tasked with proving that for each of the five crashed states, it is still possible to provide strong security guarantees about the integrity of $\widehat{L}_n$. We begin with noting that in each of the crashed states we cannot use $\sigma^{\dagger}_{n,n+1}$ for verification of L_n because it is either deleted or because the contents of L_{n+1} cannot be trusted.

This effectively means that the only way we can verify the authenticity and integrity of $\widehat{L}_n$ is with the signature of the log entry preceding it; $\sigma^{\dagger}_{n-1,n}$. In the event

that $\mathcal{A}$ deletes $\widehat{L}_n$, it needs to replace $\sigma^{\dagger}_{n-1,n}$ with a forged signature σ'_{n-1}, so that $(\widehat{L}_{n-1},\ \sigma'_{n-1})$ appears to be the last log entry. We have shown earlier that forging a signature is practically equivalent to performing a birthday attack. As such the computational complexity of performing a successful crash attack is $O(\sqrt{2^{|\mathcal{H}|}})$. □

4.4.6 *Data Corruption*

Over the course of its lifetime $\mathcal{U}$ might suffer data corruption of some parts of the log file it maintains. The purpose of this subsection is to show that corruption of any part of the log does not affect other parts of the log in terms of verifiability or decryption. Before we proceed, we take a moment to emphasize that data *corruption* and data *loss* are two similar but distinct concepts. Data corruption may lead to data loss, however depending on the severity and the nature of the corruption, this is not a given. Seeing how there has been extensive research in approaches to both mitigate [Las06] and correct [Fia+12] data corruption.

The uncorrectable bit error rate (UBER) in devices using flash storage is on the order of magnitude 10^{-15} [Mie+08]. We will now show that with the aforementioned UBER there are certain scenarios which are simply unrealistic even for the largest log entries. For these scenarios we assume ciphertexts of 2560 bits, or twenty 16 byte blocks, roughly the size of the previous paragraph if one would encrypt it using AES. The reason we use such large ciphertexts is that the probability of the scenarios described below increases with the log entry size. Therefore large ciphertexts gives us an upper bound on the likelihood for these scenarios. We assume that the signatures are 256 bits in length, a conventional MAC output digest size. And lastly we assume the UBER is constant and independent for the entire life cycle of $\mathcal{U}$.

Let the probability that a bit is erroneous be denoted as p, then we can arrive on an expression for the probability that a sequence of n bits contains *no* errors: $P(X = 0) = (1-p)^n$. This expression can then trivially be adapted to obtain the probability that a sequence of n bits contains *at least* one error: $P(X \geq 1) = 1 - P(X = 0) = 1 - (1-p)^n$. Now that we have an expression for probability that a sequence of bits will be corrupted, we demonstrate that the scenarios below are so unlikely that we do not have to account for them.

- **Corruption of both a log entry's ciphertext and signature:** The probability of this event, given the above assumptions is: $P = 1 - (1 - 10^{-15})^{2560+256} = 2.8 \times 10^{-12}$.
- **Corruption of two consecutive ciphertexts :** The likelihood of this event occurring is: $P = 1 - (1 - 10^{-15})^{2560+2560} = 5.1 \times 10^{-12}$.

By excluding the above two scenarios, we only have two possible scenarios left: either a entry's ciphertext $\widehat{L}_i$ is corrupted, or its signature $\sigma^{\dagger}_{i,i+1}$ is. In the event that the signature is corrupted, verification fails only for that particular log entry. After which it proceeds normally. In this case, $\mathcal{T}$ inspects the corresponding log entry's plaintexts; L_i and L_{i+1}. $\mathcal{T}$ determines no malicious activity has taken place and recomputes the corresponding signatures. It is at this point that $\mathcal{T}$ is able to determine that data corruption of the signature was the cause of the failed verification. In other words, despite a corrupted signature the log file is still completely verifiable, after some coordination between $\mathcal{V}$, $\mathcal{I}$ and $\mathcal{T}$.

In the event that $\widehat{L}_i$ is corrupted, then verification of $\sigma^{\dagger}_{i,i+1}$ naturally fails. Furthermore, verification of $\widehat{L}_{i-1}$ with $\sigma^{\dagger}_{i-1,i}$ and $\widehat{L}_{i+1}$ with $\sigma^{\dagger}_{i,i+1}$ fails as well. However both these ciphertexts can still be verified with $\sigma^{\dagger}_{i-2,i-1}$ and $\sigma^{\dagger}_{i+1,i+2}$ respectively. One might now think that data corruption is a viable attack vector, however from an attacker's perspective it is an extremely unreliable attack. We start with noting that the industry standard for symmetric encryption is using a block cipher such as AES in counter mode. We will therefore assume counter mode for the rest of this analysis. In counter mode, bit errors affect only the block the error has occurred in [LWR00]. Rendering that specific block unreadable, but otherwise no blocks are affected. This means that, given the extremely low UBER in flash storage and the relatively short length of ciphertexts, more than one bit error in any ciphertext is extremely unlikely and would therefore strongly indicate suspicious activity. The consequence hereof is that an attacker could potentially only corrupt a single block of the ciphertext leaving the rest of the blocks perfectly decryptable and therefore readable. For this reason we find data corruption not to be a viable attack vector, because an attacker has no way of knowing if they have erased all incriminating data.

4.5 COMPLEXITY ANALYSIS

In the title of this thesis we use the term "resource-constrained" to indicate one of our primary design objectives. To achieve this objective all IFLS operations that are performed by $\mathcal{U}$ should be efficient both in terms of computational- and space complexity. In this Section we present the complexity analysis of our scheme and evaluate whether we achieved the aforementioned "resource-constrained" goal. It is important to note however, that a theoretical analysis is only one side of the coin. The other side naturally being the performance of an implementation of the scheme on real world hardware, we address this aspect in Chapter 6.

Table 2 summarizes the complexity analysis of the various operations inherent to the IFLS scheme.

Table 2: Computational- and space complexity analysis of the IFLS scheme.

			hi-entropy-kd	*med-entropy-kd*	*lo-entropy-kd*
Computational Overhead	Initialization		$O(\lvert\mathcal{H}\rvert)$		
	Log	Key Derivation	$2 \cdot \mathcal{H}$	$\mathcal{H}$	$2 \cdot \mathtt{split}$
		Entry Creation	$2 \cdot \mathcal{H}$		
	Update		$\mathcal{H}$		
	Verify		$O(l \cdot \mathcal{H})$		
	Disclose		$O(l \cdot \mathcal{E})$		

Storage Overhead	Signer	$O(L \cdot \lvert\sigma\rvert)$
	Verifier	$O(l \cdot (\lvert\sigma\rvert + \lvert s\rvert))$
	Interpreter	$O(l \cdot (\lvert\widehat{L}\rvert + \lvert k\rvert))$

*Table 2 shows the costs associated with processing data items for each of the key derivation functions presented in this chapter. H and $\mathtt{split}$ denote the cost of doing a single hash or a split operation respectively. Note: the cost of a $\mathtt{split}$ operation will be negligible in most real world scenarios. $|H|, |\sigma|, |s|, |k|$ signifies the bit length of a hash digest, signature, a signature key and an encryption key respectively. The signing and key update costs are given for a single data item. Signature verification cost as well as item decryption cost is given for $0 < l < L$ items. The storage costs are based on the cryptographic overhead introduced by the different derivation functions, the cost of storing log events is considered to be constant and is as such omitted.

4.6 DISCUSSION

We believe that the scheme presented in this chapter is well suited for maintaining secure audit logs on offline and resource constrained devices. Compared to the other schemes evaluated in the prior art, IFLS is the only scheme which is proven to be secure against all known relevant attacks and does not require an Online TTP. In terms of computational complexity, IFLS is as fast or faster than the other solutions. Only (FI-) BAF is significantly faster with regard to number of operations needed for the **Log** and **Update** functions. (FI-) BAF requires an Online TTP however, hence we believe our solution is superior for the application scenario we have detailed earlier. In terms of space complexity (FI-) BAF, Logcrypt, FssAgg require less space than IFLS. However, the primary reason these schemes have a lower space complexity is that they communicate with a TTP. In doing so, they can transfer some of the storage cost incurred by storing data with which log entries can be verified to the TTP.

Despite its advantages, IFLS does have some flaws. These flaws are however not exclusive to just our scheme, but rather are systemic for nearly all secure logging schemes. The proofs of security presented by most secure logging schemes are correct in the theoretical sense, but fall short when translated to real world scenarios. The most serious example of hereof is in our opinion the failure to consider the limitations of persistent physical storage, especially on low-end devices.

Flash memory accounts for 34% of the annually produced semiconductor memory market [Yin08] and is predominantly used in lower-end devices, due to it being relatively cheap compared to other hard drives. However, a major limitation of flash memory is that overwriting an arbitrary number of bits is in most cases impossible to do efficiently or securely [KNM95]. The consequence of this limitation in the context of secure logging schemes is that any scheme which depends on overwrite operations in persistent storage is unsuited for use on devices with flash memory. This includes our IFLS scheme, but also a whole host of other secure logging schemes [Hol06] [MT09] [YNR12] [MT07] [YN09]. To address this problem, we present an adaptation of IFLS in Chapter 5 which sacrifices some security in return for being able to run on flash memory.

5

PSEUDORANDOM INDEXED FORWARD LINKED LOGGING

In the previous chapter we ended on the conclusion that the IFLS scheme, while secure, was not practical for most resource-constrained devices. This was due to the fact that integral to the IFLS scheme is the existence of an `override` function which efficiently and irreversibly replaces an arbitrary value in persistent storage with another value. However this is not true for low-end flash storage, because of the way these storage media are designed. On low-end flash storage, memory blocks are initialized to all zeros, then when data gets written to these blocks, the zeros are flipped to ones so that the desired bit pattern is obtained. However, these bits can exclusively be flipped from zero to one and the reverse operation is simply not possible. The only other operation that is possible, is zeroing an entire block, effectively returning it to an initialized state. Therefore, if we want to override data on flash storage, we first have to read out the entire block into memory, make the desired modifications, zero the aforementioned block and only then can we write back the modified data. This is an extremely inefficient operation and furthermore, zeroing a block in flash storage degrades the memory cells in that block quite severely, which drastically reduces the life expectancy of said block.

In this chapter we present the Pseudorandom Indexed Forward Linked logging scheme (PIFL) a novel, secure, logging scheme inspired by IFLS. PIFL is not dependent on the existence of a secure `override` operation and is therefore well-suited for use on devices with flash storage. PIFL has the same low time- and space-complexity as the IFLS scheme, it cannot however provide as strong security guarantees as IFLS provides. We expand on this claim in the security analysis section (5.2). There are two differences between PIFL and IFLS: firstly, both schemes make use of a hash chain for key derivation and the forward linked construction as described in the previous chapter. However, PIFL does not use the signature of the last entry (σ_n) as a seal. It instead keeps this signature in memory until the next log event (L_{n+1}) occurs at which point it gets updated to a forward linked signature $\sigma^{\dagger}_{n,n+1}$ and written to persistent storage along with $\widehat{L}_{n+1}$. The second difference lies in the data structure PIFL uses to store the log file. Where IFLS uses a dynamic array to which log entries get appended, PIFL uses preallocated, fixed-entry size, pseudorandomly indexed arrays. It may not be immediately obvious what these terms mean, so to elaborate:

1. **Preallocated:** PIFL reserves a fixed amount of space in persistent memory and keeps control over this space over the entire life time of $\mathcal{U}$. This space is divided up equally among A pre-allocated arrays.

2. **Fixed-entry size:** Each array entry is a fixed-, power of two, size byte block, meaning that each array entry is the same size and that log entries need to be divided up in multiple byte blocks when they are written to storage.

3. **Pseudorandomly indexed:** Using a pseudorandom function, we derive a deterministic yet unpredictable array index at which each block gets inserted.

The general idea behind PIFL is to divide a log entry into multiple blocks and insert

each block in a pseudorandom location in one of the preallocated arrays. The preallocated arrays are all equal in size. Which array an entry gets inserted in is simply a matter of starting at the first one according to some non-secret predetermined order and keep using that array until it is full. At which point, the scheme switches to the next array according to the aforementioned order. Following a predetermined order has the added benefit of making it easy to prune old log entries; we can simply zero the oldest array and use it anew, effectively simulating a ring buffer.

The reasoning behind dividing a log entry up in multiple blocks is two-fold. Firstly, an attacker that wants to delete a log entry will have to guess the indices of all associated blocks or risk certain detection. The chance of doing this correctly, assuming random guesses, approaches zero exponentially as a function of the amount of blocks in the array. Secondly, because each entry in the array is of equivalent block size, they can be stored very efficiently with virtually no overhead.

We mentioned earlier that byte blocks are power of two sized (e. g. $1, 2, 4, 8$, or 16 bytes), the reason for this restriction is simply that the output of nearly all cryptographic primitives is also power of two sized. For example, all popular cryptographic hashing algorithms; SHA256, SHA512 [PW08], Blake2 [Aum+13], SHA3 [Ber+11], MD6 [Riv+08] output either 256 or 512 bit digests. Moreover, every conventional symmetric block cipher algorithm uses 128-bit blocks. This last observation immediately places an upper bound on the maximum block size, it would be nonsensical to divide a ciphertext in blocks larger than its full-length. Seeing how no additional security would be gained and there would just be wasted space. Therefore the maximum block size is 128 bits or 16 bytes in this scheme. The trade-off one makes when deciding on what block-size to use, is that bigger block sizes mean less blocks and an attacker therefore needs to guess less-often, and the set of possible indices is smaller. Effectively making it easier to delete, or add false data to the log. However, bigger block sizes also mean less calls to the pseudorandom function and therefore a more performant scheme.

The pseudorandom function (PRF) we use to determine a block's index in the array, needs to have two requirements: **(i)** uniformity, for a set of randomly generated seeds, we want the output of the PRF to be uniformly distributed over the domain $[1, N]$ where N is the number of blocks the array can hold. **(ii)** non-correlation, a small change to the input should result in a digest that appears to be uncorrelated with the first digest. From a practical perspective, Siphash [AB12] is an ideal candidate to fulfill this role. It is a fast keyed hash function expressly designed to generate indices for hash tables and arrays. Furthermore, it is designed to be safe against "hash-flooding", a type of denial of service attack [KW11].

A problem with using a pseudorandom function to generate indices, is that collisions will occur. We address this problem by maintaining a bitmap in memory which keeps track of all the used indices. Then, in the event of a collision, we update the seed of the PRF with the generated index and generate a new index. We repeat this process until we find a non-colliding index. This approach in itself presents another problem, the number of collisions increases geometrically with the load factor (size of the array divided by the number of entries). If we were to fill each preallocated array until its load factor equals one, the performance of the scheme would degrade to an unacceptable level for inserting the last entries. We solve this problem with the

same approach hash tables use, by keeping the load factor under a certain bound. The exact value for an acceptable load factor is dependent on the computing power of $\mathcal{U}$ and the performance of the used hashing function. As a general rule of thumb, a default load factor of .75 offers a good trade-off between time and space costs [ML75].

In the previous chapter we presented a straightforward protocol for $\mathcal{U}$ to follow in the event that the IDS detects an intrusion. For PIFL we present a variation on this protocol. Like with IFLS, it involves computing an intrusion signature during the initialization phase: $\sigma_! = \mathcal{H}(\text{“}\texttt{intrusion}\text{”} \parallel s_!)$. However, unlike IFLS, we do not compute an updated signature, but rather immediately write it to randomized indices. The success probability of removing this block is akin to performing a truncation attack (5.2.1). The protocol is then as follows:

1. Delete the current signature- and encryption keys, and the hash chain node: $\texttt{delete}(Y_{n+2},\ s_{n+1},\ k_{n+1})$.
2. Compute the pseudorandom indices for the intrusion signature: $J_{|\sigma_!|}$.
3. Write the intrusion signature: $a_i[J_{|\sigma_!|}] = \sigma_!$.

If $\mathcal{A}$ interrupts the above protocol before Step 1, then he has successfully circumvented the IDS. In the case that $\mathcal{A}$ manages to stop the protocol after Step 1, then the log file is in a valid state, however $\mathcal{A}$ cannot continue logging, because the hash chain has been deleted. An interruption after Step 3 would mean that the log is in an invalid state, because there is a signature in place of where a verifier would expect a log entry to be and empty blocks where a signature should be. In this case the verifier would check whether the last log entry equals the intrusion signature, at which point the intrusion is detected.

5.1 PIFL FORMAL DEFINITION

The formal definition of PIFL resembles that of IFLS, however there are some differences. The most important of which pertains to how we define the log file $\mathcal{L}_n$. The log file in PIFL is a tuple consisting of five elements; **(i)** a set of arrays $A = [a_1, ..., a_N]$ holding the, in blocks divided, pseudorandomly indexed, log entries. **(ii)** A number S_B representing the size of the blocks in the aforementioned arrays. S_B can only be one of the values in the set $\{2^1, 2^2, 2^3, 2^4\}$ for reasons we described earlier. **(iii)** A pointer i indicating that array a_i is currently being used. **(iv)** An element representing the load factor L_f of the array a_i. And lastly **(v)** a set I containing all of the used indices in a_i. Mathematically the definition then becomes: $\mathcal{L}_n = \{A = [a_1, ..., a_N],\ S_B,\ i,\ L_f,\ I\}$.

Another difference is that PIFL instead of a single initialization log entry, takes a set of initialization entries. The reason for doing this is that if there are only few log entries in the log, the chance for an attacker to perform a successful truncation attack is relatively high. By inserting an unknown number of initialization entries in to the log, this chance becomes much smaller. Furthermore, if the attacker truncates an initialization entry, then the attack will certainly be detected upon verification.

For computing the array indices, we use a pseudorandom function for which we wield the following definition $\mathcal{F}_{\mathcal{X}}(I) \rightarrow i \in 1, ..., N, i \notin I$. It takes two parameters, a seed $\mathcal{X}$ and a set of indices I. From these two parameters it deterministically computes an index i which is *not* a member of I.

The five functions exposed by PIFL are defined as follows:

1. **Initialize** $(N_A,\ S_A,\ S_B, [L_1, ..., L_j],\ Y_1) \rightarrow \Gamma_j$ **:** First, the log file $\mathcal{L}_0$ as defined above is created, so that each array a_i can hold S_A blocks of size S_B. Then the set of initialization entries $[L_1, ..., L_j]$ and the hash chain root Y_1 are used to create the first j log entries $[(\widehat{L}_1 = \mathcal{E}_{k_1}(L_1),\ \sigma^{\dagger}_{1,2} = \mathcal{H}(\mathcal{H}(\widehat{L}_1 \| s_1)) \| \widehat{L}_2 \| s_2), ..., (\widehat{L}_j,\ \sigma_j)]$. The encryption keys $[k_1, ..., k_j]$ and the signature keys $[s_1, ..., s_j]$ are securely derived from the hash chain using one of the key derivation functions described in Section 4.3. The initialization entries are then divided up in blocks and inserted into the log file, $\mathcal{L}_0$, according to the method detailed in the **Log** function described below. The last signature σ_j is not written to the log file, but is instead kept in memory. This is due to the fact that we cannot make use of the $\mathsf{override}$ operation and as such cannot use this signature as a seal, only to compute forward-linked signatures. Finally, the initialized log state is returned: $\Gamma_j = \{\mathcal{L}_j = \{A, i, L_f, I\},\ Y_j,\ \sigma_j\}$

2. **Update** $(Y_i) \rightarrow Y_{i+1}$ **:** Evolves the current hash chain node to its next iteration and returns it.

3. **Log** $(L_i, \Gamma_{i-1}) \rightarrow \Gamma_i$ **:** Takes a new log message L_i and the previous entry's signature σ_{i-1} to create a new log entry $(\widehat{L}_i = \mathcal{E}_{k_i}(L_i),\ \sigma^{\dagger}_{i-1,i} = \mathcal{H}(\sigma_{i-1} \| \widehat{L}_i \| s_i))$. Subsequently the last signature kept in memory is updated to the signature of the last entry $\sigma_i = \mathcal{H}(\widehat{L}_i \| s_i)$. Then the log entry is divided into $N_B = \frac{|(\widehat{L}_i,\ \sigma^{\dagger}_{i-1,i})|}{S_B}$ blocks, to obtain an array of byte blocks $B = [b_1, ..., b_{N_B}]$. We then derive a seed $\mathcal{X}_i \xleftarrow{\text{derive}} Y_i$ and for each byte block b_j calculate the corresponding index as follows: $r_j = \mathcal{F}_{\mathcal{X}_i \| j}(I)$. After obtaining the index we insert the byte block in the array: $a_i[r_j] = b_j$ and update I with the new index.

4. **Verify** $([(\widehat{L}_i,\ \sigma^{\dagger}_{i,i+1}), ..., (\widehat{L}_i,\ \mathsf{null})],\ [s_i, ..., s_j]) \rightarrow \{\mathsf{valid}|\mathsf{invalid}\}$ **:** Takes a subset of log entries and a set of signature keys corresponding to that subset and mimics the logging process. Verifying the signatures at each step to check whether the log state is valid.

5. **Disclose** $([\widehat{L}_i, ..., \widehat{L}_j],\ [k_i, ..., k_j]) \rightarrow [L_i, ..., L_j]$: Takes a subset of log entries and a set of encryption keys corresponding to that subset and returns the decrypted log entries.

In Figure 5 we present an example schematic illustrating the operations performed after calling **Initialize** $(1,\ 8,\ 128,\ [L_1, L_2],\ Y_1)$. Where $\widehat{L}_1$ and $\widehat{L}_2$ are both three blocks in length and the output of $\mathcal{H}$ is 256 bits (i. e. two blocks). Note that the load factor of a_1 after this function call is 1, meaning that no more entries can be logged. This is obviously not a very useful or realistic scenario, we merely mean to provide a simple example of how our scheme functions.

Operation Description	**Persistent Storage**	**Internal State**
Initialize Arrays	$a_1 = [\varnothing, \varnothing, \varnothing, \varnothing, \varnothing, \varnothing, \varnothing, \varnothing]$ $I = \varnothing$	$Y_1 \xrightarrow{\text{Derive}} (\mathcal{X}_1, k_1, s_1)$
Create First Log Entry	$(\widehat{L}_1 = \mathcal{E}_{k_1}(L_1),\ \sigma_1 = \mathcal{H}(\widehat{L}_1 \| s_1))$	
Split Log Entry Into Blocks	$(b_1^1, b_2^1, b_3^1) = \texttt{to\_blocks}(\widehat{L}_1)$	
Compute Pseudorandom Indices	$(i_1, i_2, i_3) =$ **for** j **in** 1..3 **do:** $k = \mathcal{F}_{\mathcal{X}_1 \| j}(I)$ $I = I \cup k$ **return** k **end** $(i_1, i_2, i_3) = (2, 8, 6)$	$\texttt{Update}(Y_1)$
Update Array	$a_1 = [\varnothing, b_1^1, \varnothing, \varnothing, \varnothing, b_3^1, \varnothing, b_2^1]$ $I = [2, 6, 8]$	
Create Second Log Entry	$(\widehat{L}_2,\ \sigma_{1,2}^{\dagger} = \mathcal{H}(\sigma_1 \| \widehat{L}_2 \| s_2))$	$Y_2 \xrightarrow{\text{Derive}} (\mathcal{X}_2, k_2, s_2)$
Split Log Entry Into Blocks	$(b_1^1, b_2^1, b_3^1, b_4^1, b_5^1) = \texttt{to\_blocks}(\widehat{L}_2, \sigma_{1,2}^{\dagger})$	
Compute Pseudorandom Indices	$(i_1, i_2, i_3, i_4, i_5) = (4, 7, 1, 5, 3)$	
Update Array	$a_1 = [b_3^2, b_1^1, b_5^2, b_1^2, b_4^2, b_3^1, b_2^2, b_2^1]$ $I = [1, 2, 3, 4, 5, 6, 7, 8]$	

Figure 5: Example of the operations performed by the PIFL scheme when calling **Initialize** and **Log**.

5.2 SECURITY ANALYSIS

Due to the similarity with the IFLS scheme, most of the statements we have proven to be true in Section 4.4 are true for PIFL as well. Specifically the Building Blocks Proofs (4.4.1), the Insertion and Deletion Attacks Proofs (4.4.2) and Forging attack by Verifier or Interpreter apply to PIFL (4.4.4).

5.2.1 *Truncation Attack*

Unlike the earlier proofs presented in this thesis, the definition of "secure" against this attack is somewhat ambiguous. The ambiguity stems from the fact that the probability of executing a truncation attack successfully is a function of both the size of the log file and the number of blocks the attacker deletes. To reduce the ambiguity we start by defining a realistic scenario and making some assumptions. Then we analyze the success probabilities of a successful attack given various configurations and we try to find an optimum configuration which is both secure and realistic for light-weight devices.

For this attack we assume the worst case scenario, which is that the log contains only the initialization entries $[L_1, ..., L_j]$ and a single actual log entry L_{j+1}. $\mathcal{A}$'s goal is then to only delete all blocks associated with log entry L_{j+1}. These are the forward

linked signature blocks, $\sigma^{\dagger}_{j,j+1}$ and the blocks of the ciphertext $\widehat{L}_{j+1}$. We also assume that $\mathcal{A}$ knows the number of blocks they have to delete, the reasoning behind this assumption is that $\mathcal{A}$ could have had influence on what was being logged even before they compromised $\mathcal{U}$. Furthermore, $\mathcal{A}$ has no way of knowing which blocks to delete, only how many they have to delete. If $\mathcal{A}$ deletes a single block from one of the initialization entries, then we consider the attack failed.

Given the above scenario the problem of performing a successful truncation attack is equivalent to randomly drawing exactly one specific combination (all of the correct blocks) out of the set of all possible combinations. Mathematically, we can express this probability like: $P(\mathsf{success}) = \binom{N_{blocks}}{n_{tr}}^{-1}$. Where N_{blocks} is the total number of blocks in the log file and n_{tr} is the number of blocks $\mathcal{A}$ has to delete. Now that we have an expression for the success probability of this attack, we can determine what realistic values are for the variables in the aforementioned expression. In Chapter 6 we determined that an average line in a log file is 147 bytes in length. If we assume a block size of 16 bytes, which is the least secure size as mentioned earlier, then the ciphertext is 10 blocks in length. Subsequently, if we assume a 32 bit signature, the smallest digest size offered by the SHA2 hash family, then a log entry is in total 12 blocks in length.

We would like to emphasize that the number and size of the initialization log entries is completely under the control of $\mathcal{T}$ and different values than the ones we use can be used. For the sake of simplicity however, we assume that all of the initialization entries have the same length as the actual log entry. This means that the total number of blocks in the log file becomes $N_{blocks} = 12 \cdot (j+1)$ and correspondingly the number of blocks to truncate $n_{tr} = 12$. We find that even using only a single initialization entry ($j = 1$) already yields a very small success probability; $P(\mathsf{success}) = \binom{24}{12}^{-1} = 3.7 \times 10^{-7}$. Adding a second initialization ($j = 2$) log entry drops the probability even further to: $P(\mathsf{success}) = \binom{36}{12}^{-1} = 8.0 \times 10^{-10}$. Naturally, adding additional initialization log entries decreases the success probability even more. Figure 6 reinforces the conclusion that the chance of success approaches zero very fast for any non-small combination of block count and number of log entries.

The storage overhead created by introducing a small number of initialization log entries is negligible; a single entry costs 192 bytes and two entries 384 bytes. A cost which even the most low-end of devices can easily afford.

5.2.2 *Crash Attack*

Due to absence of an override operation in PIFL, this attack reduces to being functionally equivalent to a Truncation Attack. We first make the same Assumptions, 3 and 4, as we did for the Crash Attack Proof in Subsection 4.4.5. Then we note that there is only a single operation in the PIFL scheme which interacts with the underlying file system:

Write $(\widehat{L}_n,\ \sigma^{\dagger}_{n-1,n})$

By applying Assumptions 3, 4 and the above observation we can deduce Lemma 3

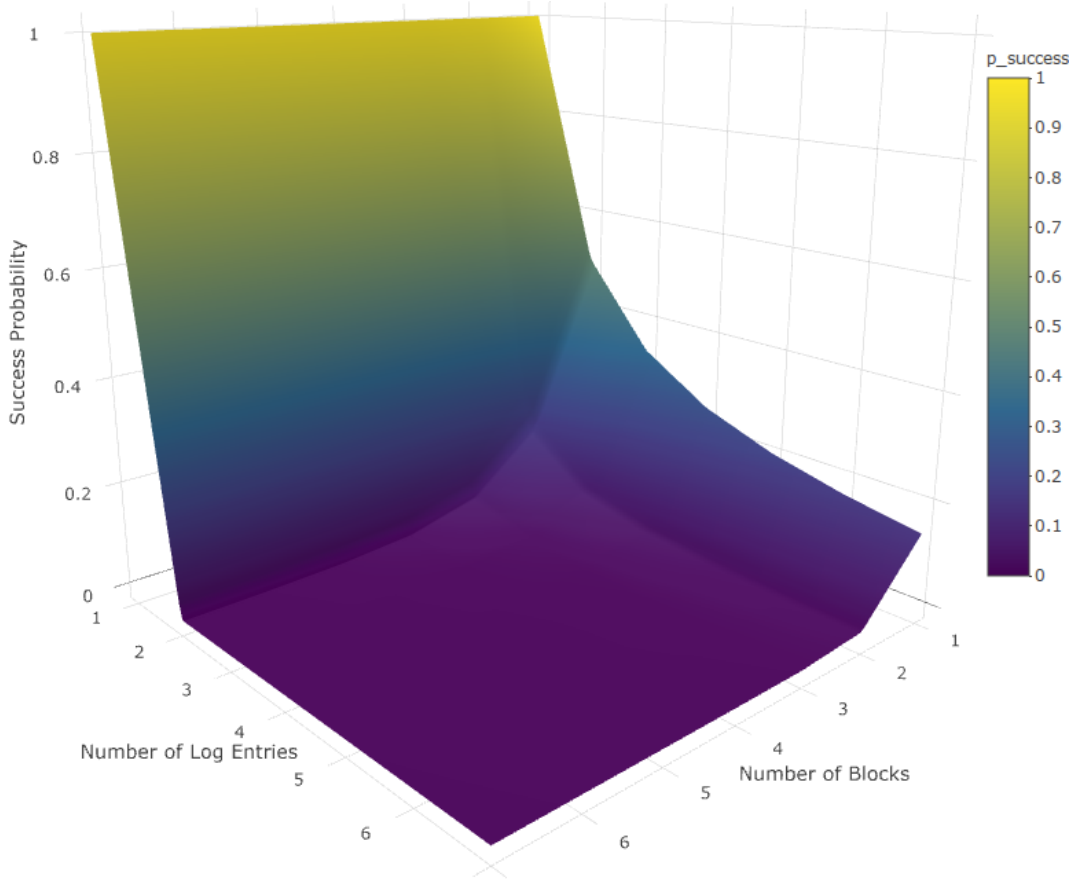


Figure 6: 3D surface plot of the success probability of performing a truncation attack, given a certain combination of block count and number of log entries.

Lemma 3. *By applying Assumption 3, 4 we conclude that from the moment $\mathcal{A}$ has gained over $\mathcal{U}$ log entry elements which are affected by file system operations are* $\widehat{L}_{n+1},\ \sigma^{\dagger}_{n,n+1}$.

To prove that our scheme is secure against the Crash Attack we assume an adversary $\mathcal{A}$ who compromises $\mathcal{U}$, deletes one or more items from the log and then crashes $\mathcal{U}$ to attempt to hide their activities. $\mathcal{A}$ is considered successful if $\mathcal{V}$ cannot determine whether the state of the log file is the result of a regular crash, or from a crash attack.

Using Assumptions 3, 4 and Lemma 3 we deduce that a regular crash can leave the log file in one of the following states:

State 1

$$\begin{aligned}\mathcal{L}_N^{Cr} = [& \quad \vdots \\ & (\widehat{L}_{n-1},\ \sigma^{\dagger}_{n-1,\,n}), \\ & (\widehat{L}_{n},\ \sigma^{\dagger}_{n,\,n+1}), \\ & (\varnothing,\ \mathtt{null})]\end{aligned}$$

State 2

$$\begin{aligned}\mathcal{L}_N^{Cr} = [\quad & \vdots \\ & (\widehat{L}_{n-1},\ \sigma^{\dagger}_{n-1,\,n}),\\ & (\widehat{L}_n,\ \varnothing),\\ & (\widehat{L}_{n+1},\ \mathtt{null})]\end{aligned}$$

State 3

$$\begin{aligned}\mathcal{L}_N^{Cr} = [\quad & \vdots \\ & (\widehat{L}_{n-1},\ \sigma^{\dagger}_{n-1,\,n}),\\ & (\widehat{L}_n,\ \varnothing),\\ & (\varnothing,\ \mathtt{null})]\end{aligned}$$

If upon verification $\mathcal{L}_N$ is in any other state than a valid state, or one of the five states listed above, then it can immediately be considered as tampered with. In this case $\mathcal{A}$ has failed in hiding its attack.

For State 1, an attacker would have to delete $\sigma^{\dagger}_{n-1,\,n}$, $\widehat{L}_n$ and $\sigma^{\dagger}_{n,\,n+1}$ to arrive at a valid state. Using the scenario and numbers from the previous section, we get a success probability of $\mathsf{P}(\mathtt{success}) = \binom{24}{14}^{-1} = 5.1 \times 10^{-7}$. In the case of state 2, an attacker would have to delete $\sigma^{\dagger}_{n-1,\,n}$, $\widehat{L}_n$ and $\widehat{L}_{n+1}$. The chance of succeeding in this case is $\mathsf{P}(\mathtt{success}) = \binom{32}{22}^{-1} = 1.6 \times 10^{-8}$. Lastly, for state 3, an attacker would need to delete $\sigma^{\dagger}_{n-1,\,n}$ and $\widehat{L}_n$, which has a chance of succeeding equal to $\mathsf{P}(\mathtt{success}) = \binom{22}{12}^{-1} = 1.6 \times 10^{-6}$. Figure 7 shows that the success probability decreases exponentially with the number of blocks in the log file. The conclusion we can draw from this analysis is that even for the most unfavorable scenario, the chance of this attack being performed successfully is incredibly small.

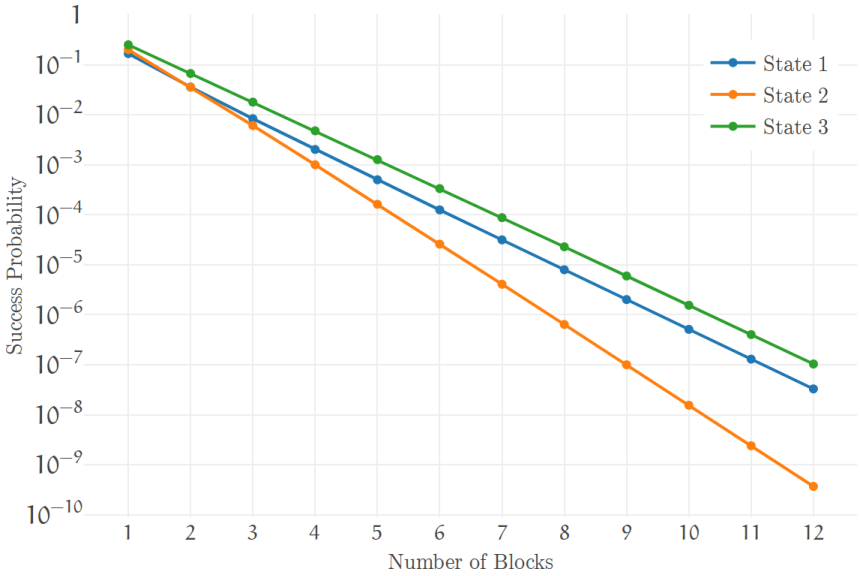


Figure 7: Line plot of the success probability of performing a successful truncation attack given a certain amount of blocks in the log file. The vertical axis is base 10 logarithmic.

5.3 COMPLEXITY ANALYSIS

Table 3 summarizes the complexity analysis of the functions that make up the PIFL scheme. The complexity of the PIFL scheme is naturally very similar to that of the IFLS scheme, as can be seen by comparing Table 2 and 3. The only real difference is that the log function has a higher computational complexity for PIFL, due to the fact that a pseudorandom index needs to be computed.

Table 3: Computational- and space complexity analysis of the IFLS scheme.

			hi-entropy-kd	*med-entropy-kd*	*lo-entropy-kd*
Computational Overhead	Initialize		$O(\lvert\mathcal{H}\rvert)$		
	Log	Key Derivation	$2 \cdot \mathcal{H}$	$\mathcal{H}$	$2 \cdot \mathrm{split}$
		Entry Creation	$2 \cdot \mathcal{H}$		
		Index Computation	$O(\lvert I\rvert)$		
	Update		$\mathcal{H}$		
	Verify		$O(l \cdot \mathcal{H})$		
	Disclose		$O(l \cdot \mathcal{E})$		

Storage Overhead	Signer	$O(L \cdot \lvert\sigma\rvert \cdot \frac{1}{L_f^{max}})$
	Verifier	$O(l \cdot (\lvert\sigma\rvert + \lvert s\rvert))$
	Interpreter	$O(l \cdot (\lvert\widehat{L}\rvert + \lvert k\rvert))$

*Table 3 shows the costs associated with processing data items for each of the key derivation functions presented in this chapter. H and split denote the cost of doing a single hash or a split operation respectively. Note: the cost of a split operation will be negligible in most real world scenarios. $|H|, |\sigma|, |s|, |k|$ signifies the bit length of a hash digest, signature, a signature key and an encryption key respectively. The signing and key update costs are given for a single data item. Signature verification cost as well as item decryption cost is given for $0 < l < L$ items. The storage costs are based on the cryptographic overhead introduced by the different derivation functions, the cost of storing log events is considered to be constant and is as such omitted.

5.4 DISCUSSION

PIFL together with the SLiC scheme by Blass and Noubir are the only schemes, as far as we know, which are secure, light-weight and suited for devices using flash storage. However, the security guarantee provided by PIFL is much stronger than that of SLiC. The probability of successfully deleting a single log entry consisting out of n blocks from a log file containing N blocks and L log entries is $\binom{N}{n}^{-1}$, while the same probability for the SLiC scheme is L^{-1}. The former probability becomes very small even for low values of n and N. In other words, PIFL offers better security and equivalent computational performance to the SLiC scheme.

A downside of the PIFL scheme is that because the arrays are only filled up until a certain load factor, there is a significant amount of allocated storage that will not be used. Naturally the amount of unused storage is a function of the maximum load factor L_f^{max}, typical values for L_f^{max} range between 0.75 and 0.95. In Subsection 6.2.1 we find that even for an L_f^{max} as high as 0.95 performance is still very fast. We realize however that this result is highly platform and implementation dependent.

6 IMPLEMENTATION

Performing just a theoretical analysis of the schemes presented in this thesis is not a strong enough basis for determining their overall quality. To be able to do this accurately we additionally need to evaluate the real world feasibility of these schemes, especially because we make the claim that our schemes are well suited for resource constrained devices. To test how well our schemes perform in realistic scenarios, we created a software implementation and benchmarked it on a high-end laptop and on a Raspberry Pi, which was chosen to represent the class of resource-constrained devices.

In this chapter we will begin with analyzing the performance of the individual building blocks of our schemes: message encryption, message authentication code computation, key derivation, updating the hash chain and writing to persistent storage. We then identify what building blocks are bottle necks for performance and why this is the case. Subsequently each of the five functions IFLS provides (`Initialize`, `Log`, `Update`, `Verify` and `Interpret`) are benchmarked. We chose to only implement the IFLS scheme because the PIFL scheme makes use of all the same building blocks as the IFLS scheme and only adds a PRF as a new component. PRFs are a well-studied phenomenon for which a great deal of implementations and benchmarks have been written. As such we deem a full implementation not essential for determining PIFL's performance characteristics, rather we will focus on just benchmarking popular PRF implementations.

We chose to write our software implementation using the Rust programming language [Hoa13]. The Rust language was created by the Mozilla Research Group, which describes it as a "safe, concurrent, practical language". Performance wise Rust is roughly on par with low level languages such as C and C++ [FG17], however both of these languages are notorious for resulting in codebases which contain hard to detect and very serious security exploits [Dur+14] [Tur14]. Rust on the contrary is safe by design; it, among other things, does not permit null pointers, data races or dangling pointers. Furthermore, Rust requires its user to manage variable lifetimes which are then reasoned about by the compiler through its *borrow checker*. We believe that Rust's performance, coupled with the safety guarantee it provides, make it an ideal language for implementing cryptographic schemes. We used Rust version 1.22.1 [NK17] for writing our implementation, the newest available version as of the time of this writing.

As mentioned earlier, we test and benchmark our scheme on two devices, a high-end laptop and a low-powered Raspberry Pi 1 Model B+ [Fou17]. The high-end laptop is equipped with a quad-core Intel i7 Broadwell processor [Int17] and 8 GB of DDR3 memory. The Raspberry Pi has a 700 MHz single core ARM11 Broadcom processor [Bro12] and 256 MB SDRAM of memory. For each of the benchmarks we perform, we run the test 100 times and measure the execution duration in nanoseconds and log size in bytes each time. Of these 100 measurements we then calculate the mean, variance and the standard deviation [Dek+05]. The raw results of all benchmarks that were run, can be found in Table 8.

6.1 IMPLEMENTATION DETAILS

The implementation we wrote contains all the functionality of the IFLS scheme as described in Chapter 4. However, how we implemented these functionalities will not be immediately obvious. In this section we will therefore detail which cryptographic primitives we used, how we translated theoretical concepts to software implementations and why we made certain design decisions. As mentioned earlier, the implementation is written in the Rust programming language. All cryptographic primitives mentioned in this section were supplied by the crate *rust-crypto 0.2.36*[1]. A *crate* is how packages are called in the Rust ecosystem. Projects can specify which crates they are dependent on and the package manager, *Cargo*, takes care of managing these dependencies. Crates are distributed through a central repository, located at `crates.io`.

It is important to note that we made a best-effort to create a secure and realistic implementation of the IFLS scheme, however this does not mean that it can withstand a capable adversary. Even though all aspects of the implementation have been tested, the code has not been professionally audited and hence it is likely that it still contains subtle bugs which can be exploited. Furthermore, side-channel attacks were not something that was considered when writing our implementation and as such, it is most likely vulnerable to these types of attacks. This implementation is instead merely meant as a starting point for any party who wants to create a truly secure software implementation of the IFLS scheme. For this reason the code has been released publicly[2] under the MIT license [Ini+06], so that it can be used with minimal restrictions by any interested party.

6.1.1 *Hash Chain and Message Authentication Codes*

Both the hash chain and the log entry signatures use the same hash family, SHA2 [PW08]. The hash chain `update` function exclusively uses the SHA512 variant of this family. The motivation for this decision was two-fold: firstly, as can be seen in Figure 8d and 8e SHA256 is not significantly faster than SHA512 and in some cases even slightly slower. Furthermore, only a single hash chain node will be in memory at any given time. The extra space gain of keeping 256 bits versus 512 bits in memory is so little that there is practically no benefit to using SHA256 in this scenario. Secondly, because the hash chain plays such a central role in our schemes, we believe that the increased security of being able to use 512 bits of entropy is always preferable over the alternative of only being able to use 256.

For the log entry signatures, HMACs are used. During the initialization phase of the scheme, the user has the option to either use SHA256 or SHA512 as the HMAC hashing function. Naturally, this choice effects the storage overhead penalty the scheme incurs. If SHA256 is chosen, each log entry signature is 256 bits in length and likewise, if SHA512 is chosen, a signature is 512 bits long. The length of the signature key s_i is 256 bits for both options.

We could have made the decision to support a larger number of hash algorithms, such as SHA3 [Ber+11], MD6 [Riv+08] or BLAKE2 [Aum+13]. However, this would

1 https://crates.io/crates/rust-crypto
2 https://github.com/p-v-d-Veeken/IFLS

not have lead to any new insights and would have resulted in a much more complex codebase. Furthermore, the purpose of this thesis is not to be a comparative study of hash algorithms, but to research novel ideas for secure logging schemes.

6.1.2 *Encryption and Decryption*

For the symmetric encryption algorithm we used AES-256 in Cipher Block Chaining (CBC) Mode [Ehr+78]. The consensus among security experts is that Counter (CTR) Mode should be the default mode of operandi when using symmetric encryption [FSK11]. We note however that in our scheme keys are not reused and as such our choice for CBC is a valid one. The initialization vector (IV) is the same for each log entry that is encrypted. The key k_i used to encrypt a log entry L_i is always 256 bits.

6.1.3 *Key Derivation*

The three key derivation methods as detailed in Section 4.3 are all implemented using the SHA2 family. The $\texttt{hi} - \texttt{entropy} - \texttt{kd}$ function concatenates the current hash chain node with the string constants "Encryption key" or "Signature key", the resulting concatenation is hashed with SHA256 to obtain the encryption key and signature key respectively. The $\texttt{med} - \texttt{entropy} - \texttt{kd}$ function concatenates the current hash chain node with the string constant "Signature & Encryption keys". The resulting concatenation is hashed using SHA512 to obtain an intermediary hash, which is then split in two equal sized parts to obtain both keys s_i, k_i. Lastly, $\texttt{lo} - \texttt{entropy} - \texttt{kd}$ splits the 512 bit hash chain node into one 256 bit part and two 128 bit parts. The two 128 bit parts are then padded to obtain the two keys. When updating the hash chain, the original, full length, hash chain node is used.

6.1.4 *Log File Structure and Encoding*

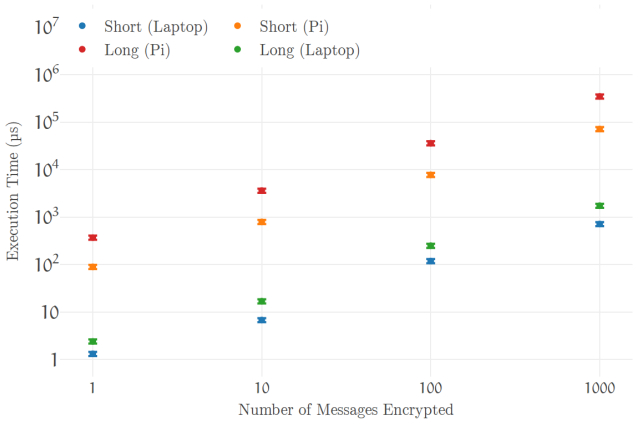
The encrypted log file is stored on the logger as a simple binary file. Whenever a new log entry is created, its ciphertext and signature are concatenated to a single byte array which is then appended to the end of the binary file. For decryption and verification purposes it is of course important to know which key derivation function was used, as well as how the signatures were computed. To this end the first two bytes of the log file are header bytes signifying the configuration which was used. In order to be able to retrieve individual entries we use length-value encoding for the log file. Length-value encoding entails that each log entry is prefaced with a 32 bit unsigned integer (i. e. 4 bytes) signifying the log entry's length. Because of the aforementioned header, the length of the entry's signature is known and hence we can recover the original, encrypted, entry $(\widehat{L}_i, \sigma_i^\dagger)$.

6.1.5 *Pseudorandom Function*

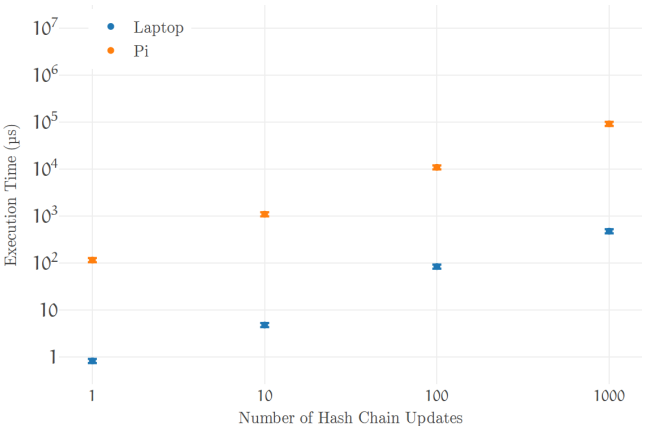
As mentioned earlier, we have not implemented the complete PIFL scheme. We have however written a benchmark for measuring the performance of using a PRF to compute indices under varying load factors. We use the SipHash [AB12] algorithm for representing the PRF. The hash function takes two parameters - a 128 bit "secret" key and an arbitrary data blob. This is different from traditional hashes which require only a data blob as an input. SipHash outputs a secure 64 bit hash.

6.2 BUILDING BLOCK BENCHMARKS

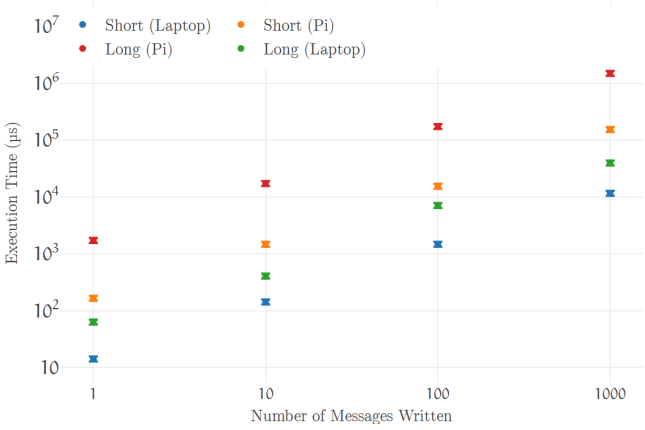
As mentioned earlier in this chapter, the five building blocks of both schemes are: **(a)** message encryption, **(b)** updating the hash chain, **(c)** writing to persistent storage, **(d)** message authentication code computation, **(e)** key derivation. To get a sense for which of these is the bottle neck in terms of performance, we benchmark each building block individually. The results hereof are displayed in Figure 8.



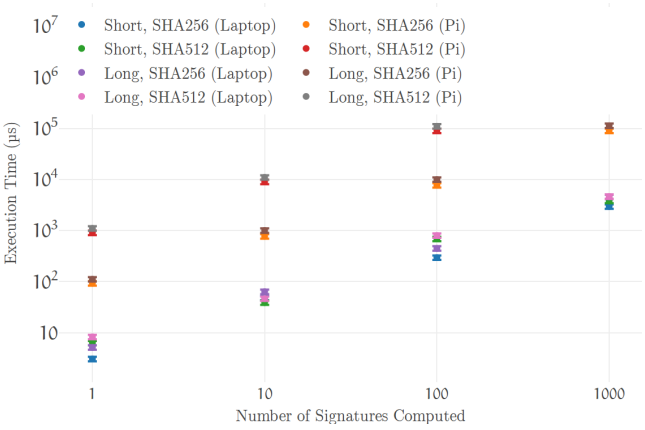
(a) Message encryption



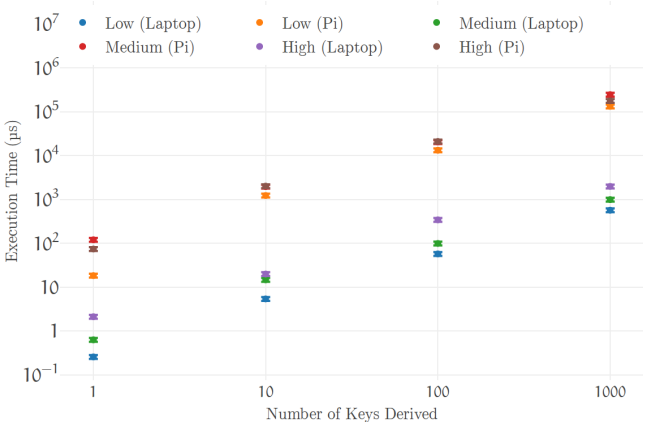
(b) Hash chain updates



(c) Writing to persistent storage



(d) Signature Computation



(e) Key derivation.

Figure 8: Scatter plots of the execution time of each of the building blocks on the high-end laptop as well as on the Raspberry Pi. The vertical axes are the execution times in µs and the horizontal axes are the number of iterations. On all of the plots, both the vertical and horizontal axes are base 10 logarithmic.

The SHA256 and SHA512 labels refer to the hash functions being used [PW08], the number coming after "SHA" designates the bit size of the output digest. For further reference see also 2.2.4. The "Short (message)" and "Long (message)" classifications refer to the length of the message being used. A short message is 15 characters in length and a long message 319 characters, respectively the maximum amount of characters than can fit in a single 128-bit block and 20 128-bit blocks. These two message sizes were not arbitrarily chosen. We took all log files, 14 in total, from the `/var/log/` directory of an often and intensively used Linux system and calculated the average log line length. This turned out to be 147 characters with the 5th and 95th percentile being 29 and 214 characters respectively. These numbers validate our choice for message lengths, because the two lengths bound the line lengths found in commonly used system logs.

From Figure 8 we conclude that both writing to persistent storage as well as signature computation are bottle necks. However, the execution time discrepancy with other building blocks is at most an order of magnitude, in the case of persistent storage versus updating the hash chain. For other comparisons the difference is typically some multiple smaller than 10. Furthermore, the key derivation functions have a much bigger impact on execution time on the laptop than on the Raspberry Pi. It is likely that the performance difference between these functions is magnified by the much faster processor of the laptop.

The Raspberry Pi appears to be around two orders of magnitude slower than the laptop for each of the building blocks, which appears to stroke with publicly available benchmarks [Ben17]. Curiously, it seems to be the case that SHA512 is as fast, or slightly faster than SHA256 on the laptop, but on the Raspberry Pi SHA512 is significantly slower than SHA256. From other benchmarks [Cry17] we gather that the expected behavior is that SHA512 is indeed slightly faster than SHA5256. However, these benchmarks are all implemented on the x64 architecture. It could therefore be possible that some factor inherent to the ARM architecture, such as the 32-bit memory addresses, are responsible for the SHA512 slow down.

6.2.1 *Pseudorandom Function Benchmarks*

In this benchmark we use the PRF implementation to pseudorandomly fill up a bitmap up until a certain load factor. As the load factor of the bitmap increases, the expected number of PRF calls that need to be done until a valid index is found, increases as well. The purpose of this benchmark is thusly to investigate how performance degrades under increasing load factors.

From Figure 9 it becomes clear that this particular implementation of a PRF is extremely fast. Even for very high load factors, the PRF is an order of magnitude faster than other building blocks. This is a promising result for the PIFL scheme, because it implies that an implementation of the scheme would be likely as fast or faster than the implementation of IFLS.

6.3 LOGGER BENCHMARKS

While decryption and verification of encrypted log files are important components of the IFLS scheme, the real acid test of our implementation is whether the logger

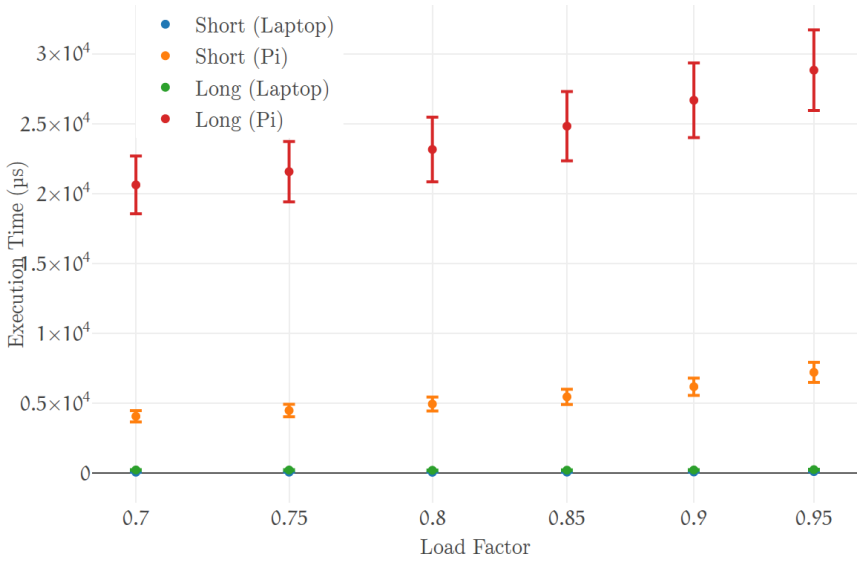


Figure 9: Scatter plot of the execution time in micro seconds of the PRF on the high-end laptop and the Raspberry Pi for increasing load factors.

performs well on the Raspberry Pi. In Figure 10 the results of the benchmarks are displayed. In this benchmark the $\texttt{med} - \texttt{entropy} - \texttt{kd}$ function was used for each of the configurations. We believe this function offers the best trade-off between performance and security, it produces keys with 256 bits of entropy and uses only a single hash function call.

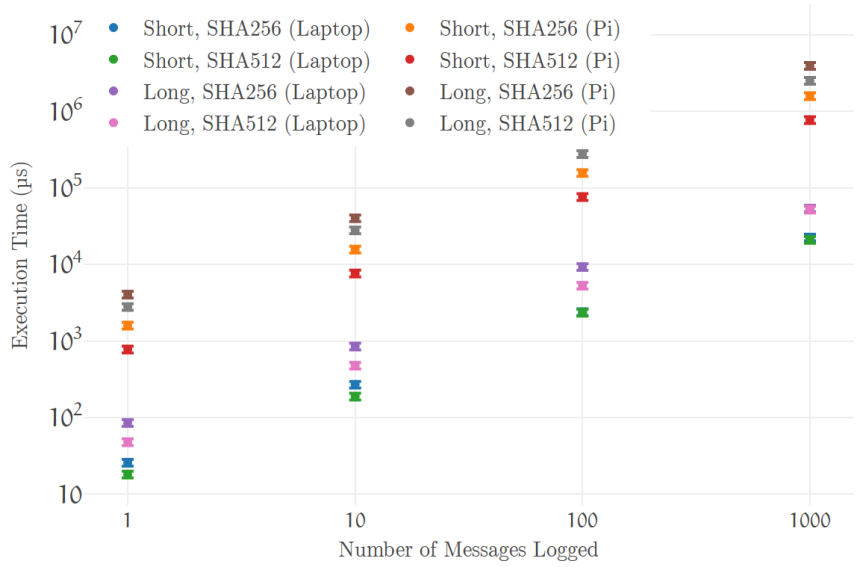


Figure 10: Scatter plot of the execution time of the logger on the high-end laptop as well as on the Raspberry Pi. Both the vertical and horizontal axes are base 10 logarithmic.

From Figure 10 we conclude that the message size is a much more dominant factor for the execution time than the choice of hashing function. This is to be expected when one considers the fact that the size of the message affects three of the five building blocks, encryption, signing and writing to the file system. Otherwise, no unexpected results present themselves in Figure 10. The Raspberry Pi is, just like in the building block benchmarks, roughly two orders of magnitude slower than the laptop. The most exciting conclusion this benchmark yields is the fact that even

in the least ideal configuration, logging only long messages and using the SHA256 algorithm, the Pi manages to log 1000 messages in 3.96 seconds. In Table 4 the performance of all configurations are presented.

Table 4: Overview of the execution time (duration) and speed (lines / s) of the logger for logging the designated number of lines using the indicated configurations on the Raspberry Pi. Each of the configurations uses the $\texttt{med} - \texttt{entropy} - \texttt{kd}$ key derivation function.

No. of Lines	Short, SHA256		Short, SHA512		Long, SHA256		Long, SHA512	
	Duration (µs)	Lines / s	Duration (µs)	Lines / s	Duration (µs)	Lines / s	Duration (µs)	Lines / s
1	1597.71	626	776.73	1287	4052.54	247	2791.12	358
10	15738.95	635	7646.75	1308	40317.18	248	27958.97	257
100	157200.89	636	76242.64	1311	407346.74	246	278769.10	359
1000	1587750.69	630	773596.45	1293	3955111.37	253	2518116.55	397

Curiously, in Figure 8 the SHA512 algorithm appears to be slower than SHA256 on the Raspberry Pi. However, in Table 4 and Figure 10 the expected behavior is demonstrated, where the SHA512 function is faster than the SHA256 function. A few probable scenarios are that the building block benchmarks somehow hit a very specific performance bottleneck, such as loss of branch prediction [Smi81] or speculative execution [CG99].

Up until this point we have only considered execution time, however the size of the log file is an important aspect of IFLS as well. In Equation 1 we define a formula for what the log file size should be for a given number of entries and a specific configuration. We arrived at this formula by combining the following observations: **(i)** the file header consists out of two bytes. **(ii)** Each log entry is preceded by a four byte integer specifying its length. **(iii)** Each log entry's cipher text consists out of either a single 128 bit block, or twenty 128 bit blocks, meaning the cipher text is either 16 or $20 \cdot 16$ bytes in length respectively. Lastly, **(iv)** depending on which hashing algorithm is used, the signature is either 32 or 64 bytes. Combining these observations, leads us to the four distinct possible combinations as defined in Equation 1.

$$\mathsf{Size}(n) = 2 + n \times \begin{cases} (4 + 1 \cdot 16 + 32) & \text{if Short, SHA256} \\ (4 + 1 \cdot 16 + 64) & \text{if Short, SHA512} \\ (4 + 20 \cdot 16 + 32) & \text{if Long, SHA256} \\ (4 + 20 \cdot 16 + 64) & \text{if Long, SHA512} \end{cases} \tag{1}$$

In Figure 11 and Table 5 the size of the log file for various configurations at increasing number of entries is presented. The recorded log file sizes follow the predicted size according to Equation 1 exactly. Which is to be expected, due to the fact that the logging benchmarks are entirely deterministic with regard to what is being logged.

From Table 5 it becomes apparent that the overhead in bytes, relative to the plain text log, decreases with the size of the message to be logged. This is to be expected, because the overhead is overwhelmingly due to the message signature, which is always of a fixed-length.

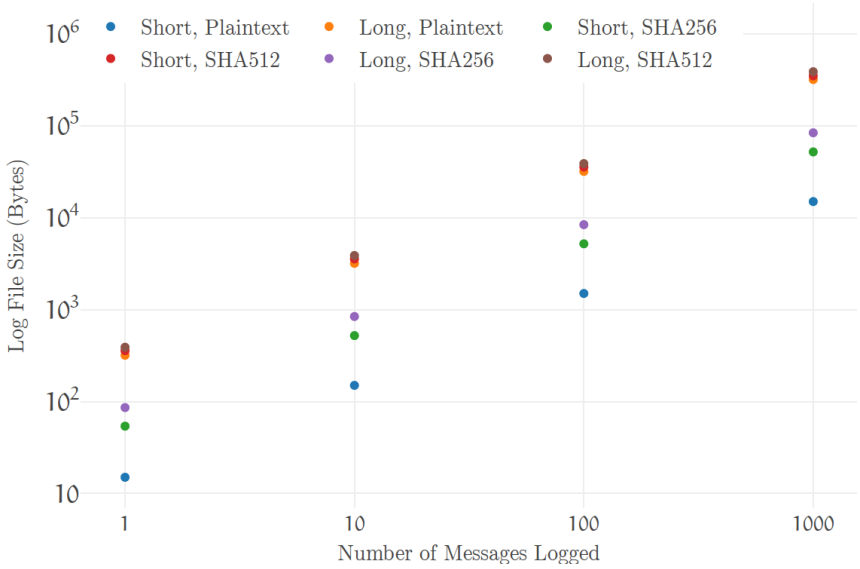


Figure 11: Scatter plot of the log file size in bytes after logging the designated number of lines. The horizontal axis is base 10 logarithmic.

Table 5: Overview of the log file size in bytes, as well as the overhead in bytes after logging the designated number of entries using the indicated configurations.

No. of Entries	Plaintext		SHA256				SHA512			
			Short		Long		Short		Long	
	Short	Long	Size	Overhead (%)	Size	Overhead (%)	Size	Overhead (%)	Size	Overhead (%)
1	15	319	54	39 (260%)	358	39 (12.2%)	86	71 (473.3%)	390	71 (22.3%)
10	150	3,190	522	372 (248%)	3,562	372 (11.7%)	842	692 (461.3%)	3,882	692 (21.7%)
100	1,500	31,900	5,202	3,702 (246.8%)	35,602	3,702 (11.6%)	8,402	6,902 (460.1%)	38,802	6,902 (21.6%)
1000	15,000	319,000	52,002	37,002 (246.7%)	356,002	37,002 (11.6%)	84,002	69,002 (460%)	388,002	69,002 (21.6%)

Now that we have size of the log file and the speed at which the Logger logs, we can look at the bandwidth of the Logger. In this case there are two kinds of bandwidth: firstly, there is the input bandwidth, which is simply the amount of plaintext bytes the Logger can process per second. The second kind is the output bandwidth, which is the number of bytes the Logger outputs per second. As we have seen with the log file size, our scheme introduces a storage overhead with each log entry. As such, the output bandwidth will be higher than the input bandwidth. In Figure 12 and Table 6 the achieved bandwidth for each of the configurations can be found.

From Figure 12 and Table 6 we conclude that the bandwidth remains fairly stable during the logging process.

6.4 INTERPRETER BENCHMARKS

The Interpreter as described in Section 4.4 is not assumed to be resource-constrained and as such its performance is less important than that of the Logger. If we however look at the operations the Interpreter needs to perform, we find no reason why it should not be able to run on a resource-constrained device. Seeing how, it merely receives an encrypted log file and a set of keys and outputs a decrypted log file. Moreover, it does not concern itself with signature computation, so at least theoreti-

Table 6: Overview of the achieved input and output bandwidths in Bytes per second for each of the denoted configurations at varying number of lines logged on the Raspberry Pi.

No. of Lines	SHA256				SHA512			
	Short		Long		Short		Long	
	Input	Output	Input	Output	Input	Output	Input	Output
1	9,388	32,547	78,716	95,742	19,312	108,146	114,291	139,012
10	9,530	31,324	79,123	95,567	19,616	106,320	114,096	137,809
100	9,542	31,189	78,312	94,521	19,674	106,279	114,432	138,118
1000	9,447	30,863	80,655	97,343	19,390	104,710	126,682	152,893

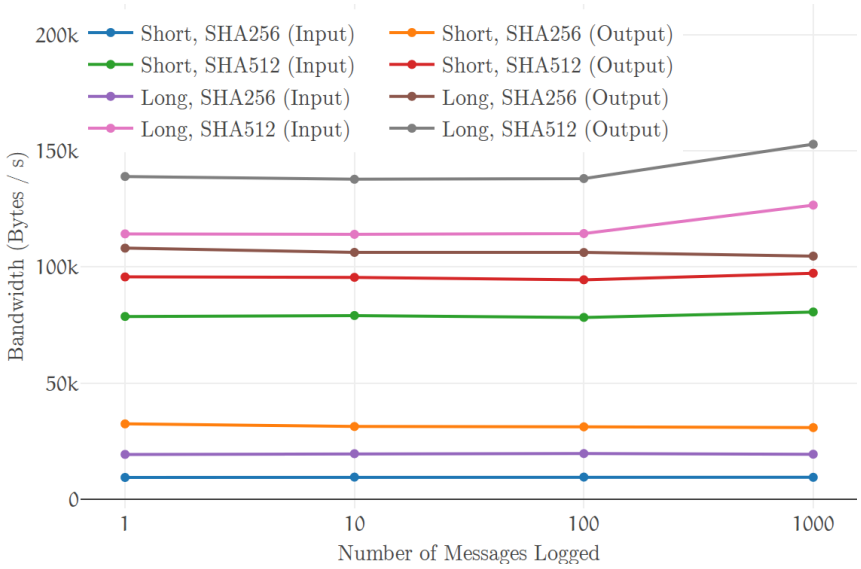


Figure 12: Line plot of the achieved input and output bandwidths for each of the denoted configurations at varying number of lines logged on the Raspberry Pi. The vertical axis is the bandwidth in Bytes per second and the horizontal axis is the number of entries logged. The horizontal axis is base 10 logarithmic.

cally it should be faster than the Logger. Under the assumption that the encryption and decryption operations are equally fast. From Figure 13 and Figure 10 we can conclude that the interpreter is indeed slightly faster than the Logger.

6.5 VERIFIER BENCHMARKS

Much like the Interpreter, the Verifier only needs to use a subset of the building blocks the Logger makes use of. Namely, signature computation and file system access. For this reason we once again expect the verifier to be slightly faster than the logger. From Figure 14 and Figure 10 we can conclude that our expectations were met and the verifier is just as performant as the Logger.

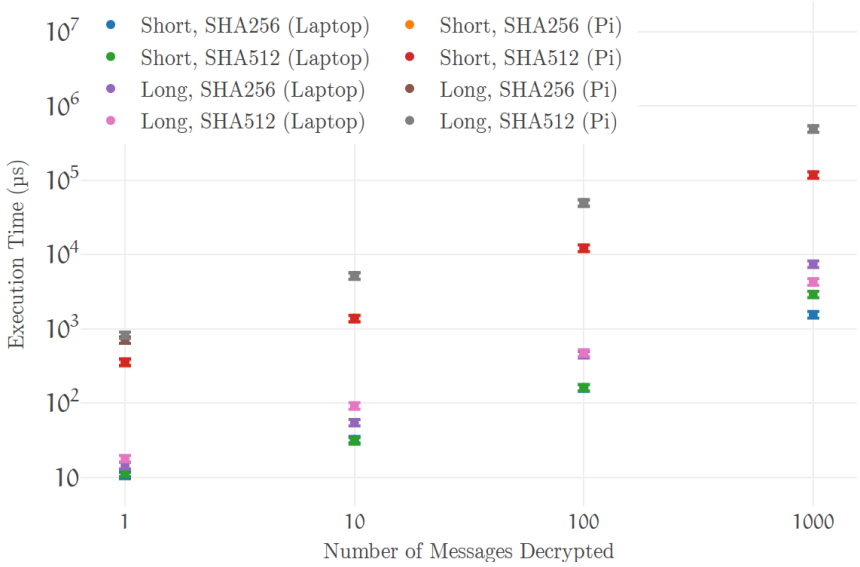


Figure 13: Scatter plot of the execution time of the interpreter on the high-end laptop as well as on the Raspberry Pi. The vertical axis is the execution time in µs and the horizontal axis is the number of iterations. Both the vertical and horizontal axes are base 10 logarithmic.

Figure 14: Scatter plot of the execution time of the verifier on the high-end laptop as well as on the Raspberry Pi. The vertical axis is the execution time in µs and the horizontal axis is the number of iterations. Both the vertical and horizontal axes are base 10 logarithmic.

6.6 CONCLUSION

In this chapter we have explicitly shown the real world feasibility of the IFLS scheme and implicitly of the PIFL scheme. Our implementation achieves writing speeds between 247 and 1311 lines per second on low-end hardware, depending on which configuration is used. It is important to note that, while performant, our implementation is relatively unoptimized. In the sense that most devices targeted by IFLS would make use of dedicated hardware for cryptographic operations such as encryption and hashing [MAD03] [SK03] [Cha+06]. These hardware implementations generally offer greatly increased speeds (and security) over simple software implementations as

used in our implementation of IFLS. Furthermore the Raspberry Pi's memory card is connected to the motherboard through a USB 2.0 adapter. This is a rather unconventional way of connecting storage to a device and is the main reason why accessing the file system on the Pi is so slow. In later editions of the Raspberry Pi, such as the model 3, the memory card is directly connected to the motherboard through a MicroSDHC slot which results in much faster file system access. These observations lead us to the conclusion that an optimized implementation could achieve writing speeds multiple times higher than what we achieved with our naive implementation.

From all the schemes discussed in Chapter 3, the only scheme which created a benchmarked implementation was SLiC from Blass and Noubir. Coincidentally they benchmark their scheme on exactly the same Raspberry Pi model as we have used for our benchmarks. Furthermore, the authors use the much of the same cryptographic primitives as we have used; AES-256 and HMAC-SHA-256. Their benchmark entails logging 2^{20} strings of 160 character each and calculating the average entries per second. This provides us with excellent data to compare our results against. In Table 7 a side-by-side comparison of achieved speeds of both schemes is given.

Table 7: Comparison of achieved speeds and bandwidths between our IFLS implementation and Blass and Noubir's SLiC implementation.

	SHA256		SHA512		
	Short	Long	Short	Long	SLiC
Speed (Lines / s)	630	247	1287	358	30
Input Bandwidth (Bytes / s)	9,447	80,655	19,390	126,682	4,800
Output Bandwidth (Bytes / s)	30,8763	97,343	104,710	152,893	8,640

From Table 7 we conclude that our implementation is superior in terms of achieved speeds to the implementation of SLiC. The differences being that IFLS is between 10 to 30 times faster than SLiC. We do note that this comparison is not perfect, due to the fact that SLiC is written in Python and makes use of an unknown PRF implementation for computing indices not present in IFLS. From our results in Subsection 6.2.1 it seems that a PRF should not result in significant performance degradation. However, without knowledge of which PRF implementation the authors used, this remains mere speculation. Python on the other hand is notorious for being slow compared to systems programming languages such as C, C++ and indeed Rust. It is therefore likely that Python is largely responsible for the perceived order of magnitude slow down.

To determine whether our scheme is truly fit for resource constrained devices one, rather arbitrary, benchmark we can look at is the throughput of a conventional serial port. Any measurement a resource-constrained device makes is most likely communicated between components through a serial port. If our scheme can match the bandwidth of such a port, we can be quite certain that our scheme satisfies the resource-constrained requirement outlined in the beginning of this thesis. A commonly used standard for serial ports is RS-232 [Eva+76]. Transmission speeds over a RS-232 port typically reach between 20,000 and 125,000 Bytes per second [Nat16]. Which, if we look at Table 6 and Figure 12, is in the range of our implementation. As mentioned earlier, this is a rather arbitrary benchmark and there are certainly other standards for serial ports in use which achieve higher throughput speeds. How-

ever, the purpose of this benchmark was to illustrate that our implementation at the very least operates on the same order of magnitude as other components of resource constrained devices.

7

DISCUSSION AND FUTURE WORK

In an ever changing IT landscape, one of the few constants have been that systems need to be audited and their actions reviewed. Log files are an essential component to the audit process and it is for this reason that secure logging schemes are important to the world of computing. There exist a great deal of secure logging schemes for an even greater amount of application scenarios. However, secure logging schemes which have been designed for resource-constrained devices are few and far between. Doubly so, for logging schemes which recognize the restrictions of the physical world and have been designed to cope with limited space and the nature of flash storage. In the literary review we conducted as part of this thesis, we could only find a single scheme, SLiC by Blass and Noubir which could realistically function on a lightweight device using flash storage.

In this chapter we revisit the research question we posed in the introductory chapter of this thesis:

> *How to construct an offline, authenticated, forward secure and confidential logging scheme which is suitable for resource-constrained devices?*

We discuss how the two schemes presented here achieve the research goal. Moreover, we provide future research directions by identifying remaining open problems and improvements.

7.1 DISCUSSION

In this thesis two secure logging schemes for resource-constrained devices are presented. We assertively state that both schemes achieve the desired properties stated in the research question; forward-security, confidentiality, authenticity and offline. The core concepts shared between IFLS and PIFL are a hash chain to derive keying material used to encrypt and sign each new log entry. And forward links established in such a manner that each log entry, safe for the last one, holds an unforgeable reference to the next entry in its signature. These forward links coupled with the hash chain derived keys ensure that any and all attack not aimed at the end of the log file is effectively mitigated. IFLS and PIFL have different strategies for counteracting attacks aimed at the tail-end of the log file.

The last log entry in a log file maintained by IFLS has a different signature than those of the preceding entries. When a new entry gets added to the log, this signature gets updated to a forward linked signature. Because the original signature cannot be retrieved after it has been updated, attacks which are aimed at the end of the log file are mitigated. Herein however also lies the crux of the IFLS scheme. The update process entails overriding the original signature with the updated one, which means IFLS relies on the file system to provide an `override` function which efficiently and irreversibly overrides a value in persistent storage. Due to the nature of flash storage, this functionality cannot be supplied by the file system on devices using this stor-

age medium. Low-end flash storage is the de-facto standard on resource-constrained devices and it is these two observations which led to the creation of the PIFL scheme.

The IFLS inspired PIFL scheme uses the hash chain and forward linked signatures the same way IFLS does. To avoid the need for an `override` function, PIFL instead divides up the log entries in blocks of 2^n bytes and pseudorandomly distributes them over preallocated arrays. Doing this makes the probability that an adversary correctly picks all the blocks associated to the last entry in the log vanishingly small.

All known attacks on secure logging schemes are computationally intractable for the IFLS scheme, assuming conventional key sizes and signature lengths. As for the PIFL scheme, most of the known attacks are computationally intractable with two specific attacks, the Truncation and Crash attacks, being statistically infeasible. With statistically infeasible we mean to say that in the worst-case scenario the probability of these attacks being successful is so small ($\sim 10^{-6}$) that we do not consider them as realistic attack vectors.

In terms of computational- and space complexity, there are a few schemes evaluated in the prior art which present better upper bounds than either IFLS and PIFL for some operations. Most notably the (FI-) BAF schemes require fewer operations for executing both the **Log** and **Update** functions. We note however that these schemes are dependent on an Online TTP and are vulnerable to some attacks.

To test the feasibility of our schemes we created a software implementation of the IFLS scheme and benchmarked the various functionalities (`log`, `verify`, `disclose` and `update`) of the scheme. Additionally, we benchmarked the building blocks used by both IFLS and PIFL. From these benchmarks we concluded that our schemes undoubtedly meets the resource-constrained requirement seeing how, without any optimizations, the IFLS implementation could match the bandwidth of a serial port on a lightweight device. Furthermore, we found that an implementation of the similar SLiC scheme was at least an order of magnitude slower than our implementation of IFLS.

We did not create a software implementations of PIFL, this does not however preclude us from drawing conclusions about its performance on low-end devices. We benchmarked most of the building blocks it uses in our IFLS benchmark, moreover we also tested the performance of a popular PRF implementation. From these benchmarks it became apparent that computing indices with the PRF, even for high load factors, was not a performance bottleneck when compared to other building blocks. Seeing how the biggest difference between IFLS and PIFL is the PRF used to compute array indices. In spite of all its advantages, we do have to place a critical note on the memory usage of PIFL. Due to the fact that $\mathcal{U}$ needs to keep track of which indices are used in the current active array, its memory usage scales linearly with the size of the array. This places an upper-bound on the maximum size of the preallocated arrays. With an efficient bitmap implementation, the memory footprint of these indices will be $|a|$ bits. Assuming arrays which can hold 2^{16} blocks, the corresponding index set will be 8192 bits (8 KB). While this memory footprint is acceptable for most devices, it is not ideal and will require frequent array switching.

7.2 FUTURE WORK

The logging schemes which we have presented in this thesis are, to the best of our knowledge, the first schemes which are provably safe against all known attacks and fit for resource-constrained devices. The two schemes, and PIFL in particular, show real potential for real-world utility in terms of security and performance. There is however still plenty of room for improvement.

7.2.1 *Finer Grained Performance Versus Security Controls*

In both the IFLS and the PIFL schemes the user can already tune the trade-off between performance and security. For both schemes there is the option to choose between one of three key derivation functions which offer varying degrees of performance versus security. Furthermore, in the PIFL scheme the user can also control the size of the blocks the log entries get divided up into. Bigger blocks mean better performance through less file system and PRF calls. The set of possible block combinations is naturally smaller however, which translates to less security against Crash and Truncation attacks.

We believe there is no one-size-fits-all solution for all the different application scenarios for even just the schemes in this thesis. As such, we deem it interesting for future research to focus on what kind of other controls can be provided with which users can control various aspects of these schemes. For example, aggregating a larger number of log entries in a forward linked signature would reduce the storage overhead at the expense of being able to verify individual log entries.

7.2.2 *Data Structures for The Log File*

The fundamental data structure underlying both IFLS and PIFL is the array. While this data structure has a lot of positive properties; performant, predictable and well-studied. It would be naive to assume that it is the single best data structure to represent a log file. Earlier research has already been conducted on this subject, however these have mainly focused on auxiliary functionalities such as creating a searchable log [Wat+04], supporting remote auditing [Acc13] or adding versioning [CW09]. During our research we have explored using binary trees to create a more corruption resistant log, however this turned out to be a dead-end. There might however be other data structures which do introduce interesting properties. Nodes in a skip list [Pug90] for example, have forward-linking to other nodes as an intrinsic property. As such, it could perhaps be used for creating a log file which is more efficient to verify, maintain or store.

Another research direction would be to understand the ramifications of representing the log file as a directed graph. What properties, for example, cycles, graph coloring, or small-world graphs introduce to the logging schemes. It could perhaps be possible to omit signatures for a subgraph of entries and still ensure that the entire log file is verifiable.

7.2.3 *Alternatives to Modifying or Hiding the Last Log Entry*

In the IFLS scheme we need a secure `override` function for persistent storage in order for the scheme to be workable. In the PIFL scheme we replace this dependence with pseudorandom indexing. The underlying reason for our need for either of these constructs, is that there needs to be someway to discern the last log entry from other log entries which is unforgeable for an outside attacker. Overriding a signature achieves this goal, pseudorandom indexing does not, but instead makes it difficult for an attacker to identify the last log entry. Another approach to achieve this goal is by using asymmetric encryption. Asymmetric encryption, as we alluded to in the introduction of this thesis, is generally computationally quite intensive. However, in their paper [YN12] Yavuz and Ning manage to make asymmetric encryption feasible for resource-constrained devices. They achieved this by using Time Released Encryption (TRE), however, which is not a concept which would work for our application scenario. The reason for this is the fact that TRE relies on online connectivity. Yavuz and Ning have nevertheless demonstrated that asymmetric encryption can be done on low-powered hardware. For this reason we believe that researching other ways light-weight asymmetric encryption can be applied to secure logging schemes is a viable research direction.

7.2.4 *Concluding Remarks*

The research objective of this thesis has been to create secure logging scheme tailored for resource-constrained devices. Although solutions have been suggested, previous work is unable to achieve resistance against all known attacks nor do they offer real-world feasibility. This is due to them making unrealistic assumptions about the underlying hardware and not considering the possibility of data corruption.

The two schemes presented in this thesis achieve the research objective by creating unforgeable links between contiguous log entries and by maintaining a hash chain to record the order of log entries. Additionally both of the schemes employ different tactics to make the last log entry unforgeable. The first scheme achieves this property by making use of updating signatures so that the signature of the last entry cannot be retrieved after it has been updated. The second scheme makes the last log entry unforgeable by randomizing the positions of all the entries in the log, so that without knowledge of the seeds of the PRF, an attacker cannot identify the last log entry.

The performance and security achieved by both schemes presented in this thesis is, to the best of our knowledge, unparalleled in the context of resource-constrained devices. Moreover we believe that a secure and efficient implementation of the PIFL scheme would be a viable option for a large number of non-trivial use cases. Our implementation of the IFLS scheme would in this case be an excellent starting point to base this secure implementation of.

APPENDIX

A

BENCHMARK RESULTS

Table 8: Results of all the benchmarks run on the Laptop and the Raspberry Pi. The durations are in µs and for each result the standard deviation (σ) is reported.

	Duration (µs ± σ)	
Benchmark	Laptop	Pi
calc_1_sigs_msg_len_15_sha256	3 ± 0.148	$93 \pm 4.55 \times 10^6$
calc_1_sigs_msg_len_15_sha512	6 ± 1.36	$905 \pm 4.28 \times 10^6$
calc_1_sigs_msg_len_319_sha256	5 ± 1.45	$111 \pm 3.27 \times 10^5$
calc_1_sigs_msg_len_319_sha512	8 ± 1.97	$1,102 \pm 4.02 \times 10^6$
calc_10_sigs_msg_len_15_sha256	48 ± 7.04	$771 \pm 2.61 \times 10^6$
calc_10_sigs_msg_len_15_sha512	39 ± 2.01	$8,972 \pm 5.38 \times 10^7$
calc_10_sigs_msg_len_319_sha256	63 ± 12.1	$1,001 \pm 7.38 \times 10^6$
calc_10_sigs_msg_len_319_sha512	47 ± 3.45	$10,993 \pm 4.07 \times 10^7$
calc_100_sigs_msg_len_15_sha256	295 ± 8.31	$7,681 \pm 8.32 \times 10^6$
calc_100_sigs_msg_len_15_sha512	684 ± 134	$89,838 \pm 3.04 \times 10^8$
calc_100_sigs_msg_len_319_sha256	448 ± 8.12	$9,961 \pm 4.77 \times 10^7$
calc_100_sigs_msg_len_319_sha512	800 ± 76.8	$109,993 \pm 3.25 \times 10^8$
calc_1000_sigs_msg_len_15_sha256	$2,950 \pm 51.8$	$89,705 \pm 2.72 \times 10^8$
calc_1000_sigs_msg_len_15_sha512	$3,803 \pm 40.5$	$753,041 \pm 7.86 \times 10^8$
calc_1000_sigs_msg_len_319_sha256	$4,575 \pm 283$	$112,806 \pm 2.57 \times 10^8$
calc_1000_sigs_msg_len_319_sha512	$4,636 \pm 220$	$924,428 \pm 3.35 \times 10^9$
decrypt_1_messages_msg_len_15	1 ± 2.46	$84 \pm 4.95 \times 10^6$
decrypt_1_messages_msg_len_319	2 ± 0.32	$400 \pm 4.36 \times 10^5$
decrypt_10_messages_msg_len_15	8 ± 1.32	$726 \pm 2.17 \times 10^6$
decrypt_10_messages_msg_len_319	16 ± 0.197	$3,954 \pm 8.91 \times 10^6$
decrypt_100_messages_msg_len_15	75 ± 14.3	$7,049 \pm 1.37 \times 10^7$
decrypt_100_messages_msg_len_319	159 ± 4.66	$39,361 \pm 1.87 \times 10^8$
decrypt_1000_messages_msg_len_15	878 ± 179	$70,560 \pm 2.27 \times 10^8$
decrypt_1000_messages_msg_len_319	$1,714 \pm 269$	$395,862 \pm 3.4 \times 10^{12}$
derive_1_keys_High	2 ± 0.382	$74 \pm 3.61 \times 10^5$
derive_1_keys_Low	0 ± 0.059	$18 \pm 1.91 \times 10^5$
derive_1_keys_Medium	1 ± 0.126	$120 \pm 2.18 \times 10^5$
derive_10_keys_High	20 ± 2.42	$2,022 \pm .37 \times 10^8$
derive_10_keys_Low	5 ± 0.064	$1,231 \pm 9.55 \times 10^6$
derive_10_keys_Medium	15 ± 5.82	$1,970 \pm 5.78 \times 10^6$
derive_100_keys_High	344 ± 35.1	$21,146 \pm 1.39 \times 10^8$
derive_100_keys_Low	57 ± 3.93	$13,320 \pm 7.43 \times 10^7$
derive_100_keys_Medium	99 ± 3.88	$20,518 \pm 6.53 \times 10^7$
derive_1000_keys_High	$1,992 \pm 211$	$178,442 \pm 6.38 \times 10^8$
derive_1000_keys_Low	570 ± 9.72	$133,778 \pm 5.52 \times 10^8$
derive_1000_keys_Medium	995 ± 14.6	$247,417 \pm 4.63 \times 10^9$
do_1_hashchain_updates	1 ± 0.283	$116 \pm 3.24 \times 10^5$
do_10_hashchain_updates	5 ± 1.12	$1,099 \pm .47 \times 10^7$
do_100_hashchain_updates	84 ± 7.75	$10,920 \pm 5.87 \times 10^7$
do_1000_hashchain_updates	477 ± 77.7	$91,848 \pm 2.29 \times 10^8$
encrypt_1_messages_msg_len_15	1 ± 1.99	$90 \pm .39 \times 10^6$

	Duration ($\mu s \pm \sigma$)	
Benchmark	Laptop	Pi
encrypt_1_messages_msg_len_319	2 ± 0.374	$373 \pm 2.33 \times 10^6$
encrypt_10_messages_msg_len_15	7 ± 0.573	$798 \pm 4.17 \times 10^6$
encrypt_10_messages_msg_len_319	17 ± 0.49	$3,641 \pm 3.27 \times 10^7$
encrypt_100_messages_msg_len_15	119 ± 58.8	$7,843 \pm .27 \times 10^8$
encrypt_100_messages_msg_len_319	251 ± 107	$36,359 \pm 2.16 \times 10^8$
encrypt_1000_messages_msg_len_15	719 ± 123	$72,235 \pm 2.74 \times 10^8$
encrypt_1000_messages_msg_len_319	$1,749 \pm 230$	$351,390 \pm .23 \times 10^{10}$
fill_to_load_factor_0.7_entry_len_22	$230 \pm 2,280$	$20,618 \pm 4.21 \times 10^{13}$
fill_to_load_factor_0.7_entry_len_3	68 ± 672	$4,066 \pm 1.64 \times 10^{12}$
fill_to_load_factor_0.75_entry_len_22	$225 \pm 2,240$	$21,565 \pm 4.60 \times 10^{13}$
fill_to_load_factor_0.75_entry_len_3	71 ± 708	$4,482 \pm 1.99 \times 10^{12}$
fill_to_load_factor_0.8_entry_len_22	$205 \pm 2,040$	$23,153 \pm 5.31 \times 10^{13}$
fill_to_load_factor_0.8_entry_len_3	63 ± 630	$4,944 \pm 2.42 \times 10^{12}$
fill_to_load_factor_0.85_entry_len_22	$214 \pm 2,130$	$24,820 \pm 6.10 \times 10^{13}$
fill_to_load_factor_0.85_entry_len_3	75 ± 750	$5,461 \pm 2.95 \times 10^{12}$
fill_to_load_factor_0.9_entry_len_22	$226 \pm 2,250$	$26,673 \pm 7.04 \times 10^{13}$
fill_to_load_factor_0.9_entry_len_3	83 ± 829	$6,185 \pm 3.79 \times 10^{12}$
fill_to_load_factor_0.95_entry_len_22	$253 \pm 2,520$	$28,824 \pm 8.22 \times 10^{13}$
fill_to_load_factor_0.95_entry_len_3	$111 \pm 1,100$	$7,213 \pm 5.15 \times 10^{12}$
interpret_1_lines_block_len_1_high_sha256	12 ± 71.6	$357 \pm 4.31 \times 10^7$
interpret_1_lines_block_len_1_high_sha512	7 ± 37.1	$361 \pm .46 \times 10^8$
interpret_1_lines_block_len_1_low_sha256	11 ± 72.9	$348 \pm .39 \times 10^8$
interpret_1_lines_block_len_1_low_sha512	11 ± 73.7	$357 \pm 4.31 \times 10^7$
interpret_1_lines_block_len_1_med_sha256	17 ± 132	$359 \pm 4.43 \times 10^7$
interpret_1_lines_block_len_1_med_sha512	7 ± 38.2	$368 \pm 4.43 \times 10^7$
interpret_1_lines_block_len_20_high_sha256	15 ± 115	$713 \pm 3.19 \times 10^8$
interpret_1_lines_block_len_20_high_sha512	11 ± 72.7	$719 \pm .32 \times 10^9$
interpret_1_lines_block_len_20_low_sha256	14 ± 109	$706 \pm 3.15 \times 10^8$
interpret_1_lines_block_len_20_low_sha512	18 ± 140	$823 \pm 4.53 \times 10^8$
interpret_1_lines_block_len_20_med_sha256	19 ± 150	$709 \pm 3.14 \times 10^8$
interpret_1_lines_block_len_20_med_sha512	15 ± 115	$712 \pm .32 \times 10^9$
interpret_10_lines_block_len_1_high_sha256	36 ± 315	$1,368 \pm 1.48 \times 10^9$
interpret_10_lines_block_len_1_high_sha512	20 ± 164	$1,432 \pm 1.49 \times 10^9$
interpret_10_lines_block_len_1_low_sha256	32 ± 285	$1,394 \pm 1.54 \times 10^9$
interpret_10_lines_block_len_1_low_sha512	31 ± 270	$1,370 \pm .15 \times 10^{10}$
interpret_10_lines_block_len_1_med_sha256	30 ± 264	$1,442 \pm 1.65 \times 10^9$
interpret_10_lines_block_len_1_med_sha512	23 ± 199	$1,376 \pm 1.52 \times 10^9$
interpret_10_lines_block_len_20_high_sha256	91 ± 860	$5,231 \pm 2.56 \times 10^{10}$
interpret_10_lines_block_len_20_high_sha512	49 ± 454	$5,013 \pm 2.34 \times 10^{10}$

	Duration ($\mu s \pm \sigma$)	
Benchmark	Laptop	Pi
interpret__10__lines__block__len__20__low__sha256	54 ± 509	$5,172 \pm 2.5 \times 10^{10}$
interpret__10__lines__block__len__20__low__sha512	91 ± 870	$5,125 \pm 2.46 \times 10^{10}$
interpret__10__lines__block__len__20__med__sha256	87 ± 829	$5,233 \pm 2.57 \times 10^{10}$
interpret__10__lines__block__len__20__med__sha512	54 ± 506	$5,129 \pm 2.46 \times 10^{10}$
interpret__100__lines__block__len__1__high__sha256	$273 \pm 2,680$	$12,129 \pm 1.42 \times 10^{11}$
interpret__100__lines__block__len__1__high__sha512	$154 \pm 1,500$	$12,008 \pm 1.39 \times 10^{11}$
interpret__100__lines__block__len__1__low__sha256	$159 \pm 1,550$	$12,147 \pm 1.42 \times 10^{11}$
interpret__100__lines__block__len__1__low__sha512	$162 \pm 1,580$	$12,226 \pm 1.44 \times 10^{11}$
interpret__100__lines__block__len__1__med__sha256	$272 \pm 2,670$	$11,946 \pm 1.38 \times 10^{11}$
interpret__100__lines__block__len__1__med__sha512	$158 \pm 1,540$	$11,920 \pm 1.36 \times 10^{11}$
interpret__100__lines__block__len__20__high__sha256	$721 \pm 7,130$	$49,946 \pm 2.46 \times 10^{12}$
interpret__100__lines__block__len__20__high__sha512	$425 \pm 4,200$	$49,951 \pm 2.45 \times 10^{12}$
interpret__100__lines__block__len__20__low__sha256	$447 \pm 4,420$	$49,361 \pm .24 \times 10^{13}$
interpret__100__lines__block__len__20__low__sha512	$473 \pm 4,680$	$50,179 \pm 2.48 \times 10^{12}$
interpret__100__lines__block__len__20__med__sha256	$638 \pm 6,300$	$49,571 \pm 2.42 \times 10^{12}$
interpret__100__lines__block__len__20__med__sha512	$439 \pm 4,330$	$50,031 \pm 2.46 \times 10^{12}$
interpret__1000__lines__block__len__1__high__sha256	$2,194 \pm 21,800$	$118,066 \pm 1.38 \times 10^{13}$
interpret__1000__lines__block__len__1__high__sha512	$2,055 \pm 20,400$	$119,392 \pm 1.41 \times 10^{13}$
interpret__1000__lines__block__len__1__low__sha256	$1,551 \pm 15,400$	$117,898 \pm 1.37 \times 10^{13}$
interpret__1000__lines__block__len__1__low__sha512	$2,903 \pm 28,900$	$118,511 \pm 1.39 \times 10^{13}$
interpret__1000__lines__block__len__1__med__sha256	$1,810 \pm 18,000$	$118,154 \pm 1.38 \times 10^{13}$
interpret__1000__lines__block__len__1__med__sha512	$1,512 \pm 15,000$	$118,403 \pm 1.38 \times 10^{13}$
interpret__1000__lines__block__len__20__high__sha256	$4,520 \pm 44,900$	$494,117 \pm 2.42 \times 10^{13}$
interpret__1000__lines__block__len__20__high__sha512	$5,448 \pm 54,200$	$492,476 \pm 2.40 \times 10^{14}$
interpret__1000__lines__block__len__20__low__sha256	$7,437 \pm 74,000$	$493,615 \pm 2.41 \times 10^{13}$
interpret__1000__lines__block__len__20__low__sha512	$4,291 \pm 42,700$	$492,868 \pm 2.40 \times 10^{14}$
interpret__1000__lines__block__len__20__med__sha256	$4,862 \pm 48,300$	$494,067 \pm 2.42 \times 10^{13}$
interpret__1000__lines__block__len__20__med__sha512	$4,389 \pm 43,600$	$493,119 \pm 2.41 \times 10^{13}$
log__1__lines__block__len__1__high__sha256	17 ± 1.64	$1,478 \pm 1.23 \times 10^{7}$
log__1__lines__block__len__1__high__sha512	17 ± 3.12	$714 \pm 8.37 \times 10^{6}$
log__1__lines__block__len__1__low__sha256	26 ± 5.78	$1,598 \pm 2.34 \times 10^{7}$
log__1__lines__block__len__1__low__sha512	18 ± 2.12	$777 \pm 1.16 \times 10^{7}$
log__1__lines__block__len__1__med__sha256	18 ± 2.22	$1,339 \pm 7.78 \times 10^{6}$
log__1__lines__block__len__1__med__sha512	18 ± 4.33	$703 \pm 6.93 \times 10^{6}$
log__1__lines__block__len__20__high__sha256	46 ± 5.74	$3,929 \pm 6.59 \times 10^{7}$
log__1__lines__block__len__20__high__sha512	50 ± 10.7	$2,717 \pm 5.68 \times 10^{7}$
log__1__lines__block__len__20__low__sha256	85 ± 24.2	$4,053 \pm 4.82 \times 10^{7}$
log__1__lines__block__len__20__low__sha512	48 ± 3.81	$2,791 \pm 3.65 \times 10^{7}$
log__1__lines__block__len__20__med__sha256	48 ± 7.65	$3,470 \pm 2.35 \times 10^{7}$
log__1__lines__block__len__20__med__sha512	50 ± 12.2	$2,423 \pm 2.19 \times 10^{7}$
log__1__lines__only__io__msg__len__15	14 ± 2.45	$165 \pm 1.09 \times 10^{6}$
log__1__lines__only__io__msg__len__319	63 ± 15.1	$1,720 \pm 1.97 \times 10^{7}$

	Duration ($\mu s \pm \sigma$)	
Benchmark	Laptop	Pi
log_10_lines_block_len_1_high_sha256	234 ± 41.9	$14,766 \pm 1.36 \times 10^{8}$
log_10_lines_block_len_1_high_sha512	171 ± 23.4	$7,022 \pm 7.31 \times 10^{7}$
log_10_lines_block_len_1_low_sha256	269 ± 51.3	$15,739 \pm 9.54 \times 10^{7}$
log_10_lines_block_len_1_low_sha512	188 ± 31.4	$7,647 \pm 6.21 \times 10^{7}$
log_10_lines_block_len_1_med_sha256	252 ± 40.3	$13,324 \pm 6.07 \times 10^{7}$
log_10_lines_block_len_1_med_sha512	182 ± 34.7	$6,914 \pm 4.05 \times 10^{7}$
log_10_lines_block_len_20_high_sha256	464 ± 27.8	$39,345 \pm 5.53 \times 10^{8}$
log_10_lines_block_len_20_high_sha512	466 ± 31.1	$26,678 \pm 1.17 \times 10^{8}$
log_10_lines_block_len_20_low_sha256	849 ± 257	$40,317 \pm 6.24 \times 10^{8}$
log_10_lines_block_len_20_low_sha512	478 ± 28.7	$27,959 \pm 2.21 \times 10^{8}$
log_10_lines_block_len_20_med_sha256	485 ± 73.9	$39,403 \pm 2.33 \times 10^{12}$
log_10_lines_block_len_20_med_sha512	545 ± 130	$24,555 \pm 1.02 \times 10^{8}$
log_10_lines_only_io_msg_len_15	143 ± 13.2	$1,477 \pm 1.85 \times 10^{7}$
log_10_lines_only_io_msg_len_319	407 ± 101	$17,182 \pm 9.47 \times 10^{7}$
log_100_lines_block_len_1_high_sha256	$2,356 \pm 665$	$144,389 \pm 3.56 \times 10^{9}$
log_100_lines_block_len_1_high_sha512	$2,140 \pm 742$	$69,364 \pm 7.95 \times 10^{8}$
log_100_lines_block_len_1_low_sha256	$2,389 \pm 557$	$157,201 \pm 2.42 \times 10^{9}$
log_100_lines_block_len_1_low_sha512	$2,347 \pm 730$	$76,243 \pm 1.02 \times 10^{9}$
log_100_lines_block_len_1_med_sha256	$2,430 \pm 705$	$133,499 \pm 2.14 \times 10^{9}$
log_100_lines_block_len_1_med_sha512	$2,382 \pm 646$	$68,767 \pm 1.06 \times 10^{9}$
log_100_lines_block_len_20_high_sha256	$5,887 \pm 1,550$	$391,669 \pm 5.65 \times 10^{9}$
log_100_lines_block_len_20_high_sha512	$5,197 \pm 1,220$	$277,246 \pm .46 \times 10^{13}$
log_100_lines_block_len_20_low_sha256	$9,297 \pm 1,510$	$407,347 \pm 1.28 \times 10^{12}$
log_100_lines_block_len_20_low_sha512	$5,299 \pm 1,140$	$278,769 \pm 3.67 \times 10^{10}$
log_100_lines_block_len_20_med_sha256	$5,565 \pm 1,290$	$356,166 \pm 7.79 \times 10^{12}$
log_100_lines_block_len_20_med_sha512	$5,534 \pm 1,360$	$250,232 \pm 3.26 \times 10^{12}$
log_100_lines_only_io_msg_len_15	$1,478 \pm 83.2$	$15,404 \pm 8.26 \times 10^{7}$
log_100_lines_only_io_msg_len_319	$7,066 \pm 1,350$	$173,064 \pm 1.66 \times 10^{9}$
log_1000_lines_block_len_1_high_sha256	$18,955 \pm 2,410$	$1,448,381 \pm 1.63 \times 10^{13}$
log_1000_lines_block_len_1_high_sha512	$18,487 \pm 2,870$	$701,943 \pm 1.14 \times 10^{10}$
log_1000_lines_block_len_1_low_sha256	$22,615 \pm 5,040$	$1,587,751 \pm 5.51 \times 10^{12}$
log_1000_lines_block_len_1_low_sha512	$21,054 \pm 2,570$	$773,596 \pm 1.27 \times 10^{13}$
log_1000_lines_block_len_1_med_sha256	$20,445 \pm 3,030$	$1,339,440 \pm 2.32 \times 10^{12}$
log_1000_lines_block_len_1_med_sha512	$18,305 \pm 2,190$	$708,512 \pm 1.54 \times 10^{13}$
log_1000_lines_block_len_20_high_sha256	$52,167 \pm 6,920$	$3,361,878 \pm 6.54 \times 10^{12}$
log_1000_lines_block_len_20_high_sha512	$51,542 \pm 8,570$	$2,720,546 \pm 7.51 \times 10^{10}$
log_1000_lines_block_len_20_low_sha256	$54,048 \pm 8,890$	$3,955,111 \pm 4.10 \times 10^{13}$
log_1000_lines_block_len_20_low_sha512	$52,370 \pm 7,350$	$2,518,117 \pm 3.36 \times 10^{13}$
log_1000_lines_block_len_20_med_sha256	$49,648 \pm 4,940$	$3,437,183 \pm 5.21 \times 10^{10}$
log_1000_lines_block_len_20_med_sha512	$50,272 \pm 7,300$	$2,455,094 \pm 5.99 \times 10^{10}$
log_1000_lines_only_io_msg_len_15	$11,580 \pm 1,230$	$153,035 \pm .12 \times 10^{10}$
log_1000_lines_only_io_msg_len_319	$39,450 \pm 5,660$	$1,492,572 \pm 3.47 \times 10^{12}$

	Duration ($\mu s \pm \sigma$)	
Benchmark	Laptop	Pi
verify_1_lines_block_len_1_high_sha256	5 ± 0.524	$209 \pm 1.84 \times 10^6$
verify_1_lines_block_len_1_high_sha512	4 ± 1.37	$202 \pm 6.84 \times 10^5$
verify_1_lines_block_len_1_low_sha256	4 ± 1.3	$206 \pm 1.02 \times 10^6$
verify_1_lines_block_len_1_low_sha512	3 ± 1.29	$201 \pm 4.02 \times 10^5$
verify_1_lines_block_len_1_med_sha256	3 ± 0.905	$205 \pm 1.02 \times 10^6$
verify_1_lines_block_len_1_med_sha512	5 ± 1.08	$202 \pm .4 \times 10^6$
verify_1_lines_block_len_20_high_sha256	6 ± 0.459	$534 \pm 1.28 \times 10^6$
verify_1_lines_block_len_20_high_sha512	4 ± 1.41	$535 \pm 1.54 \times 10^6$
verify_1_lines_block_len_20_low_sha256	6 ± 1.53	$536 \pm 2.16 \times 10^6$
verify_1_lines_block_len_20_low_sha512	4 ± 0.524	$538 \pm 2.37 \times 10^6$
verify_1_lines_block_len_20_med_sha256	5 ± 1.96	$535 \pm 1.28 \times 10^6$
verify_1_lines_block_len_20_med_sha512	4 ± 0.392	$539 \pm 2.66 \times 10^6$
verify_10_lines_block_len_1_high_sha256	87 ± 2.02	$7,518 \pm .15 \times 10^8$
verify_10_lines_block_len_1_high_sha512	37 ± 5.48	$1,409 \pm 4.78 \times 10^6$
verify_10_lines_block_len_1_low_sha256	53 ± 13.2	$7,527 \pm 1.54 \times 10^7$
verify_10_lines_block_len_1_low_sha512	37 ± 2.75	$1,430 \pm 5.01 \times 10^6$
verify_10_lines_block_len_1_med_sha256	74 ± 18.7	$7,511 \pm 1.43 \times 10^7$
verify_10_lines_block_len_1_med_sha512	38 ± 6.36	$1,435 \pm 4.76 \times 10^6$
verify_10_lines_block_len_20_high_sha256	127 ± 8.4	$11,718 \pm 6.83 \times 10^7$
verify_10_lines_block_len_20_high_sha512	99 ± 24.7	$2,171 \pm 1.21 \times 10^7$
verify_10_lines_block_len_20_low_sha256	93 ± 24.7	$11,710 \pm 6.22 \times 10^7$
verify_10_lines_block_len_20_low_sha512	66 ± 2.12	$2,216 \pm 8.07 \times 10^7$
verify_10_lines_block_len_20_med_sha256	71 ± 15.9	$11,719 \pm .82 \times 10^8$
verify_10_lines_block_len_20_med_sha512	68 ± 7.23	$2,192 \pm 1.06 \times 10^7$
verify_100_lines_block_len_1_high_sha256	856 ± 123	$79,747 \pm 4.04 \times 10^8$
verify_100_lines_block_len_1_high_sha512	371 ± 20.5	$13,092 \pm 7.09 \times 10^7$
verify_100_lines_block_len_1_low_sha256	695 ± 156	$79,666 \pm 3.16 \times 10^8$
verify_100_lines_block_len_1_low_sha512	362 ± 18.8	$13,110 \pm .81 \times 10^8$
verify_100_lines_block_len_1_med_sha256	469 ± 33	$79,686 \pm 2.92 \times 10^8$
verify_100_lines_block_len_1_med_sha512	387 ± 75.2	$13,110 \pm 3.83 \times 10^7$
verify_100_lines_block_len_20_high_sha256	$1,107 \pm 268$	$122,176 \pm 3.87 \times 10^8$
verify_100_lines_block_len_20_high_sha512	714 ± 102	$18,063 \pm 6.88 \times 10^7$
verify_100_lines_block_len_20_low_sha256	822 ± 182	$122,136 \pm 5.78 \times 10^8$
verify_100_lines_block_len_20_low_sha512	738 ± 147	$18,077 \pm 7.34 \times 10^7$
verify_100_lines_block_len_20_med_sha256	673 ± 37	$122,166 \pm .49 \times 10^9$
verify_100_lines_block_len_20_med_sha512	844 ± 238	$18,033 \pm 2.94 \times 10^7$
verify_1000_lines_block_len_1_high_sha256	$5,144 \pm 930$	$800,095 \pm 2.23 \times 10^9$
verify_1000_lines_block_len_1_high_sha512	$4,163 \pm 954$	$129,671 \pm 3.55 \times 10^8$
verify_1000_lines_block_len_1_low_sha256	$5,006 \pm 766$	$800,723 \pm 1.97 \times 10^9$
verify_1000_lines_block_len_1_low_sha512	$4,136 \pm 867$	$131,402 \pm 8.42 \times 10^8$
verify_1000_lines_block_len_1_med_sha256	$5,036 \pm 909$	$800,296 \pm 1.85 \times 10^9$
verify_1000_lines_block_len_1_med_sha512	$3,949 \pm 807$	$129,536 \pm 3.63 \times 10^8$
verify_1000_lines_block_len_20_high_sha256	$7,134 \pm 1,080$	$1,226,005 \pm 3.63 \times 10^9$
verify_1000_lines_block_len_20_high_sha512	$7,316 \pm 1,220$	$176,707 \pm .75 \times 10^9$
verify_1000_lines_block_len_20_low_sha256	$7,174 \pm 926$	$1,227,331 \pm 3.01 \times 10^{11}$
verify_1000_lines_block_len_20_low_sha512	$7,104 \pm 748$	$178,311 \pm 6.62 \times 10^8$
verify_1000_lines_block_len_20_med_sha256	$7,835 \pm 2,220$	$1,226,107 \pm 3.49 \times 10^9$
verify_1000_lines_block_len_20_med_sha512	$7,258 \pm 1,060$	$176,733 \pm 1.12 \times 10^9$

BIBLIOGRAPHY

[AR00] Michel Abdalla and Leonid Reyzin. "A New Forward-Secure Digital Signature Scheme." In: *Advances in Cryptology — ASIACRYPT 2000: 6th International Conference on the Theory and Application of Cryptology and Information Security Kyoto, Japan, December 3–7, 2000 Proceedings*. Ed. by Tatsuaki Okamoto. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 116–129. ISBN: 978-3-540-44448-0. DOI: 10.1007/3-540-44448-3_10. URL: https://doi.org/10.1007/3-540-44448-3_10.

[Acc13] Rafael Accorsi. "A secure log architecture to support remote auditing." In: *Mathematical and Computer Modelling* 57.7 (2013). Public Key Services and Infrastructures EUROPKI-2010-Mathematical Modelling in Engineering & Human Behaviour 2011, pp. 1578–1591. ISSN: 0895-7177. DOI: http://dx.doi.org/10.1016/j.mcm.2012.06.035. URL: http://www.sciencedirect.com/science/article/pii/S0895717712001744.

[Adi08] Ben Adida. "Helios: Web-based Open-Audit Voting." In: *USENIX security symposium*. Vol. 17. 2008, pp. 335–348.

[AKV03] Michael Alles, Alexander Kogan, and Miklos Vasarhelyi. "Black box logging and tertiary monitoring of continuous assurance systems." In: *Information Systems Control Journal* 1 (2003), pp. 37–41.

[Asi20] Isaac Asimov. *I, Asimov: a Memoir*. Bantam Doubleday Dell Publishing Group Inc, Jan. 1, 1920. ISBN: 055356997X.

[AL10] Yonatan Aumann and Yehuda Lindell. "Security Against Covert Adversaries: Efficient Protocols for Realistic Adversaries." In: *J. Cryptol.* 23.2 (Apr. 2010), pp. 281–343. ISSN: 0933-2790. DOI: 10.1007/s00145-009-9040-7. URL: http://dx.doi.org/10.1007/s00145-009-9040-7.

[AB12] Jean-Philippe Aumasson and Daniel J Bernstein. "SipHash: A Fast Short-Input PRF." In: *INDOCRYPT*. Vol. 7668. Springer. 2012, pp. 489–508.

[Aum+13] Jean-Philippe Aumasson et al. "BLAKE2: simpler, smaller, fast as MD5." In: *International Conference on Applied Cryptography and Network Security*. Springer. 2013, pp. 119–135.

[Bel06] Mihir Bellare. "New proofs for NMAC and HMAC: Security without collision-resistance." In: *Annual International Cryptology Conference*. Springer. 2006, pp. 602–619.

[Bel17] Mihir Bellare. *CSE 107 – Introduction to Modern Cryptography*. Lecture. Jan. 2017. URL: https://cseweb.ucsd.edu/~mihir/cse107/index.html.

[BK04] Mihir Bellare and Tadayoshi Kohno. "Hash function balance and its impact on birthday attacks." In: *International Conference on the Theory and Applications of Cryptographic Techniques*. Springer. 2004, pp. 401–418.

[BM99] Mihir Bellare and Sara K. Miner. "A Forward-Secure Digital Signature Scheme." In: *Proceedings of the 19th Annual International Cryptology Conference on Advances in Cryptology*. CRYPTO '99. London, UK, UK: Springer-Verlag, 1999, pp. 431–448. ISBN: 3-540-66347-9. URL: http://dl.acm.org/citation.cfm?id=646764.703986.

[Ben17] CPU Benchmarks. "PassMark Software." In: *URL: http://www. cpubenchmark. net/singleThread. html, updated June* 2 (2017).

[Ber09] Daniel J. Bernstein. "Cost analysis of hash collisions: Will quantum computers make SHARCS obsolete?" In: *SHARCS*. 2009.

[Ber+11] Guido Bertoni et al. "The keccak sha-3 submission." In: *Submission to NIST (Round 3)* 6.7 (2011), p. 16.

[BN17] Erik-Oliver Blass and Guevara Noubir. *Secure Logging with Crash Tolerance*. Cryptology ePrint Archive, Report 2017/107. http://eprint.iacr.org/2017/107. 2017.

[Bro12] BroadCom. "Broadcom.com - BCM2835." In: *URL http://www.broadcom.com/products/BCM2835* (2012).

[BDH11] Johannes Buchmann, Erik Dahmen, and Andreas Hülsing. "XMSS - a Practical Forward Secure Signature Scheme Based on Minimal Security Assumptions." In: *Proceedings of the 4th International Conference on Post-Quantum Cryptography*. PQCrypto'11. Taipei, Taiwan: Springer-Verlag, 2011, pp. 117–129. ISBN: 978-3-642-25404-8. DOI: 10.1007/978-3-642-25405-5_8. URL: http://dx.doi.org/10.1007/978-3-642-25405-5_8.

[CGH04] Ran Canetti, Oded Goldreich, and Shai Halevi. "The Random Oracle Methodology, Revisited." In: *J. ACM* 51.4 (July 2004), pp. 557–594. ISSN: 0004-5411. DOI: 10.1145/1008731.1008734. URL: http://doi.acm.org/10.1145/1008731.1008734.

[CC03] Jae Choon Cha and Jung Hee Cheon. "An Identity-Based Signature from Gap Diffie-Hellman Groups." In: *Proceedings of the 6th International Workshop on Theory and Practice in Public Key Cryptography: Public Key Cryptography*. PKC '03. London, UK, UK: Springer-Verlag, 2003, pp. 18–30. ISBN: 3-540-00324-X. URL: http://dl.acm.org/citation.cfm?id=648120.746918.

[CHS07] Konstantinos Chalkias, Dimitrios Hristu-Varsakelis, and George Stephanides. "Improved anonymous timed-release encryption." In: *Computer Security–ESORICS 2007* (2007), pp. 311–326.

[CG99] Fay Chang and Garth Gibson. "Automatic I/O hint generation through speculative execution." In: *Proc. 3rd Symp. Operating System Design and Implementation*. USENIX. 1999.

[Cha+06] Ricardo Chaves et al. "Improving SHA-2 hardware implementations." In: *International Workshop on Cryptographic Hardware and Embedded Systems*. Springer. 2006, pp. 298–310.

[CP10] Anton Chuvakin and Gunnar Peterson. "How to Do Application Logging Right." In: *IEEE Security and Privacy* 8.4 (July 2010), pp. 82–85. ISSN: 1540-7993. DOI: 10.1109/MSP.2010.127. URL: http://dx.doi.org/10.1109/MSP.2010.127.

[CW09] Scott A. Crosby and Dan S. Wallach. "Efficient Data Structures for Tamper-evident Logging." In: *Proceedings of the 18th Conference on USENIX Security Symposium*. SSYM'09. Montreal, Canada: USENIX Association, 2009, pp. 317–334. URL: http://dl.acm.org/citation.cfm?id=1855768.1855788.

[Cry17] Dai W Crypto+. "Speed Comparison of Popular Crypto Algorithms." In: *URL http://www. cryptopp. com/benchmarks. html* (2017).

[Dek+05] F. M. Dekking et al. *A Modern Introduction to Probability and Statistics*. Springer London Ltd, May 6, 2005. 488 pp. ISBN: 1852338962.

[DVW92] Whitfield Diffie, Paul C. Van Oorschot, and Michael J. Wiener. "Authentication and authenticated key exchanges." In: *Designs, Codes and Cryptography* 2.2 (June 1992), pp. 107–125. ISSN: 1573-7586. DOI: 10.1007/BF00124891. URL: https://doi.org/10.1007/BF00124891.

[Dow+16] Benjamin Dowling et al. *Secure Logging Schemes and Certificate Transparency*. Cryptology ePrint Archive, Report 2016/452. http://eprint.iacr.org/2016/452. 2016.

[Dur+14] Zakir Durumeric et al. "The matter of heartbleed." In: *Proceedings of the 2014 Conference on Internet Measurement Conference*. ACM. 2014, pp. 475–488.

[EKK14] Mansoor Ebrahim, Shujaat Khan, and Umer Bin Khalid. "Symmetric Algorithm Survey: A Comparative Analysis." In: *International Journal of Computer Applications 61.20 (2013)* (May 2, 2014). arXiv: 1405.0398v1 [cs.CR].

[Ehr+78] William F Ehrsam et al. *Message verification and transmission error detection by block chaining*. US Patent 4,074,066. Feb. 1978.

[Esh07] Tamir Eshel. *LANdroid Robots to Support Communications in Urban Combat*. Blog Post. Accessed: 14 August 2017. June 2007.

[Eva+76] John M Evans Jr et al. *Standards for Computer Aided Manufacturing*. Tech. rep. NATIONAL BUREAU OF STANDARDS WASHINGTON DC INST FOR COMPUTER SCIENCES and TECHNOLOGY, 1976.

[Fef10] William Jason Fefferman. "On quantum computing and pseudorandomness." PhD thesis. California Institute of Technology, 2010.

[FSK11] Niels Ferguson, Bruce Schneier, and Tadayoshi Kohno. *Cryptography engineering: design principles and practical applications*. John Wiley & Sons, 2011.

[Fia+12] David Fiala et al. "Detection and correction of silent data corruption for large-scale high-performance computing." In: *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE Computer Society Press. 2012, p. 78.

[Fou17] Raspberry Pi Foundation. "Raspberry Pi 1 Model B+ - Raspberry Pi." In: *URL https://www.raspberrypi.org/products/raspberry-pi-1-model-b/* (2017).

[FG17] Brent Fulgham and Isaac Gouy. "The computer language benchmarks game." In: *available from (accessed 2017-12-5)* (2017).

[Gol09] Oded Goldreich. *Foundations of cryptography: volume 2, basic applications*. Cambridge university press, 2009.

[GMR88] Shafi Goldwasser, Silvio Micali, and Ronald L Rivest. "A digital signature scheme secure against adaptive chosen-message attacks." In: *SIAM Journal on Computing* 17.2 (1988), pp. 281–308.

[HS91] Stuart Haber and W. Scott Stornetta. "How to Time-Stamp a Digital Document." In: *Proceedings of the 10th Annual International Cryptology Conference on Advances in Cryptology*. CRYPTO '90. London, UK, UK: Springer-Verlag, 1991, pp. 437–455. ISBN: 3-540-54508-5. URL: http://dl.acm.org/citation.cfm?id=646755.705358.

[Hoa13] G Hoare. "The rust programming language." In: *URL http://www. rust-lang. org* (2013).

[Hol06] Jason E. Holt. "Logcrypt: Forward Security and Public Verification for Secure Audit Logs." In: *Proceedings of the 2006 Australasian Workshops on Grid Computing and e-Research - Volume 54*. ACSW Frontiers '06. Hobart, Tasmania, Australia: Australian Computer Society, Inc., 2006, pp. 203–211. ISBN: 1-920-68236-8. URL: http://dl.acm.org/citation.cfm?id=1151828.1151852.

[Ini+06] Open Source Initiative et al. *The MIT license*. 2006.

[Int17] Intel. "Intel Core i7-5600U Processor (4M Cache, up to 3.20 GHz) Product Specifications." In: *URL https://ark.intel.com/products/85215/* (2017).

[KNM95] Atsuo Kawaguchi, Shingo Nishioka, and Hiroshi Motoda. "A Flash-Memory Based File System." In: *USENIX*. 1995, pp. 155–164.

[KS06] Karen Kent and Murugiah Souppaya. "Guide to computer security log management." In: *NIST special publication* 800.92 (2006), pp. 16–16.

[KD06] Jack Kerouac and Anne Douglas. *The Dharma Bums*. Penguin Books Ltd, Nov. 1, 2006. 224 pp. ISBN: 0143039601. URL: http://www.ebook.de/de/product/5384554/jack_kerouac_anne_douglas_the_dharma_bums.html.

[KW11] Alexander Klink and Julian Walde. "Efficient Denial of Service Attacks on Web Application Platforms." In: *The 28th Chaos Communication Congress*. 2011.

[Knu68] Donald E. Knuth. *Art of Computer Programming, Volume 1: Fundamental Algorithms*. Addison-Wesley Educational Publishers Inc, 1968. ISBN: 0-201-03801-3.

[KCB97] Hugo Krawczyk, Ran Canetti, and Mihir Bellare. "HMAC: Keyed-hashing for message authentication." In: *Network Working Group* (1997).

[Las06] Menahem Lasser. *Flash memory management method that is resistant to data corruption by power loss*. US Patent 6,988,175. Jan. 2006.

[LWR00] Helger Lipmaa, David Wagner, and Phillip Rogaway. "Comments to NIST concerning AES modes of operation: CTR-mode encryption." In: *Citeseer* (2000).

[Ma08] Di Ma. "Practical Forward Secure Sequential Aggregate Signatures." In: *Proceedings of the 2008 ACM Symposium on Information, Computer and Communications Security*. ASIACCS '08. Tokyo, Japan: ACM, 2008, pp. 341–352. ISBN: 978-1-59593-979-1. DOI: 10.1145/1368310.1368361. URL: http://doi.acm.org/10.1145/1368310.1368361.

[MT07] Di Ma and Gene Tsudik. *Forward-Secure Sequential Aggregate Authentication*. June 2007.

[MT09] Di Ma and Gene Tsudik. "A New Approach to Secure Logging." In: *Trans. Storage* 5.1 (Mar. 2009), 2:1–2:21. ISSN: 1553-3077. DOI: 10.1145/1502777.1502779. URL: http://doi.acm.org/10.1145/1502777.1502779.

[Maa04] Martijn Maas. "Pairing-based cryptography." In: *Master's thesis, Technische Universiteit Eindhoven* (2004).

[MMM02] Tal Malkin, Daniele Micciancio, and Sara Miner. "Efficient generic forward-secure signatures with an unbounded number of time periods." In: *Advances in Cryptology - Eurocrypt 2002*. Vol. 2332. Lectture Notes in Computer Science. IACR. Amsterdam, The Netherlands: Springer-Verlag, Apr. 2002, pp. 400–417.

[MAD03] Stefan Mangard, Manfred Aigner, and Sandra Dominikus. "A highly regular and scalable AES hardware architecture." In: *IEEE Transactions on Computers* 52.4 (2003), pp. 483–491.

[ML75] Ward Douglas Maurer and Theodore Gyle Lewis. "Hash table methods." In: *ACM Computing Surveys (CSUR)* 7.1 (1975), pp. 5–19.

[MOV96] Alfred John Menezes, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography*. Taylor & Francis Inc, Oct. 16, 1996. 810 pp. ISBN: 0849385237.

[Mer87] Ralph C Merkle. "A digital signature based on a conventional encryption function." In: *Conference on the Theory and Application of Cryptographic Techniques*. Springer. 1987, pp. 369–378.

[Mie+08] Neal Mielke et al. "Bit error rate in NAND flash memories." In: *Reliability Physics Symposium, 2008. IRPS 2008. IEEE International*. IEEE. 2008, pp. 9–19.

[NM02] Junko Nakajima and Mitsuru Matsui. "Performance analysis and parallel implementation of dedicated hash functions." In: *Advances in Cryptology – EUROCRYPT 2002*. Springer. 2002, pp. 165–180.

[Nat16] National-Instruments. "RS-232, RS-422, RS-485 Serial Communication General Concepts." In: *URL: http://www.ni.com/white-paper/11390/en/* (2016).

[NK17] Alex Chricton Nick Cameron and Yehuda Katz. "Announcing Rust 1.22 (and 1.22.1)." In: *URL https://blog.rust-lang.org/2017/11/22/Rust-1.22.html* (2017).

[PW08] Wouter Penard and Tim van Werkhoven. "On the secure hash algorithm family." In: *National Security Agency, Tech. Rep.* (2008).

[Pug90] William Pugh. "Skip Lists: A Probabilistic Alternative to Balanced Trees." In: *Commun. ACM* 33.6 (June 1990), pp. 668–676. ISSN: 0001-0782. DOI: 10.1145/78973.78977. URL: http://doi.acm.org/10.1145/78973.78977.

[RP01] B Rees and P Prando. "Documentation and log keeping: ensuring your work does what you intend it to do." In: *Health physics* 81.3 (2001), pp. 265–268.

[Riv+08] Ronald L Rivest et al. "The MD6 hash function–a proposal to NIST for SHA-3." In: *Submission to NIST* 2.3 (2008).

[SK98] Bruce Schneier and John Kelsey. "Cryptographic Support for Secure Logs on Untrusted Machines." In: *Proceedings of the 7th Conference on USENIX Security Symposium - Volume 7*. SSYM'98. San Antonio, Texas: USENIX Association, 1998, pp. 4–4. URL: http://dl.acm.org/citation.cfm?id=1267549.1267553.

[Sha01] Claude E Shannon. "A mathematical theory of communication." In: *ACM SIGMOBILE Mobile Computing and Communications Review* 5.1 (2001), pp. 3–55.

[SK03] Nicolas Sklavos and Odysseas Koufopavlou. "On the hardware implementations of the SHA-2 (256, 384, 512) hash functions." In: *Circuits and Systems, 2003. ISCAS'03. Proceedings of the 2003 International Symposium on*. Vol. 5. IEEE. 2003, pp. V–V.

[Sma15] Nigel Smart. *Cryptography Made Simple*. Springer-Verlag GmbH, Dec. 14, 2015. ISBN: 3319219359.

[Smi81] James E Smith. "A study of branch prediction strategies." In: *Proceedings of the 8th annual symposium on Computer Architecture*. IEEE Computer Society Press. 1981, pp. 135–148.

[Sti05] Douglas Stinson. *Cryptography*. Taylor & Francis Ltd., Oct. 11, 2005. 593 pp. ISBN: 1584885084.

[SA08] NR Sunitha and BB Amberker. "Some aggregate forward-secure signature schemes." In: *TENCON 2008-2008 IEEE Region 10 Conference*. IEEE. 2008, pp. 1–6.

[Tur14] Stephen Turner. "Security vulnerabilities of the top ten programming languages: C, Java, C++, Objective-C, C#, PHP, Visual Basic, Python, Perl, and Ruby." In: *Journal of Technology Research* 5 (2014), p. 1.

[Wat+04] Brent R Waters et al. "Building an Encrypted and Searchable Audit Log." In: *NDSS*. Vol. 4. 2004, pp. 5–6.

[YNR12] Attila A. Yavuz, Peng Ning, and Michael K. Reiter. "BAF and FI-BAF: Efficient and Publicly Verifiable Cryptographic Schemes for Secure Logging in Resource-Constrained Systems." In: *ACM Trans. Inf. Syst. Secur.* 15.2 (July 2012), 9:1–9:28. ISSN: 1094-9224. DOI: 10.1145/2240276.2240280. URL: http://doi.acm.org/10.1145/2240276.2240280.

[YN09] Attila Altay Yavuz and Peng Ning. "BAF: An Efficient Publicly Verifiable Secure Audit Logging Scheme for Distributed Systems." In: *2009 Annual Computer Security Applications Conference* (2009), pp. 219–228.

[YN12] Attila Altay Yavuz and Peng Ning. "Self-sustaining, Efficient and Forward-secure Cryptographic Constructions for Unattended Wireless Sensor Networks." In: *Ad Hoc Netw.* 10.7 (Sept. 2012), pp. 1204–1220. ISSN: 1570-8705. DOI: 10.1016/j.adhoc.2012.03.006. URL: http://dx.doi.org/10.1016/j.adhoc.2012.03.006.

[Yin08] Falan Yinug. "The rise of the flash memory market: Its impact on firm behavior and global semiconductor trade patterns." In: *J. Int'l Com. & Econ.* 1 (2008), p. 137.