Thesis no: MECS-2014-10

Improving Integrity Assurances of Log Entries

From the Perspective of Intermittently Disconnected Devices

Marcus Andersson
Alexander Nilsson

Faculty of Computing
Blekinge Institute of Technology
SE–371 79 Karlskrona, Sweden

This thesis is submitted to the Faculty of Computing at Blekinge Institute of Technology in partial fulfillment of the requirements for the degree of Master of Science in Engineering. The thesis is equivalent to 20 weeks of full-time studies.

Contact Information:
Author(s):
Marcus Andersson
E-mail: mban09@student.bth.se
Alexander Nilsson
E-mail: alnb09@student.bth.se

University advisor:
Dr. Stefan Axelsson
Dept. Computer Science & Engineering

External advisor:
Peter Bayer
DinGard AB

Faculty of Computing
Blekinge Institute of Technology
SE–371 79 Karlskrona, Sweden

Internet : www.bth.se
Phone : +46 455 38 50 00
Fax : +46 455 38 50 57

Abstract

Context. It is common today in large corporate environments for system administrators to employ centralized systems for log collection and analysis. The log data can come from any device between smart-phones and large scale server clusters. During an investigation of a system failure or suspected intrusion these logs may contain vital information. However, the trustworthiness of this log data must be confirmed.

Objectives. The objective of this thesis is to evaluate the state of the art and provide practical solutions and suggestions in the field of secure logging. In this thesis we focus on solutions that do not require a persistent connection to a central log management system.

Methods. To this end a prototype logging framework was developed including client, server and verifier applications. The client employs different techniques of signing log entries. The focus of this thesis is to evaluate each signing technique from both a security and performance perspective.

Results. This thesis evaluates "Traditional RSA-signing", "Traditional Hash-chains", "Itkis-Reyzin's asymmetric FSS scheme" and "RSA signing and tick-stamping with TPM", the latter being a novel technique developed by us. In our evaluations we recognized the inability of the evaluated techniques to detect so called 'truncation-attacks', therefore a truncation detection module was also developed which can be used independent of and side-by-side with any signing technique.

Conclusions. In this thesis we conclude that our novel "RSA signing and tick-stamping with TPM" technique has the most to offer in terms of log security, however it does introduce a hardware dependency on the *Trusted Platform Module*. We have also shown that the truncation detection technique can be used to assure an external verifier of the number of log entries that has *at least* passed through the log client software.

Keywords: Secure logging, forward security, TPM, digital signature

List of Figures

3.1 Overview of forward secure signing methods 16
3.2 "Itkis-Reyzin's asymmetric FSS scheme" key generation algorithm 18
3.3 "Itkis-Reyzin's asymmetric FSS scheme" key update algorithm . . 18
3.4 "Itkis-Reyzin's asymmetric FSS scheme" signing algorithm 19
3.5 "Itkis-Reyzin's asymmetric FSS scheme" verification algorithm . . 19
3.6 Overview of the truncation detection technique 22

5.1 Overview of architecture 27
5.2 Excerpts from the TPM specification (TPM_SIGN_INFO) . . . 34
5.3 Excerpt from the TPM specification (TSS_VALIDATION) 34

List of Tables

3.1 Traditional RSA-signing 15
3.2 Traditional Hash-chains 15
3.3 Itkis-Reyzin's asymmetric FSS scheme 17
3.4 RSA signing and tick-stamping with TPM 20

5.1 Table of currently implemented pipeline stages. 28
5.2 Performance results: Signing 37
5.3 Performance results: Setup & key update 38
5.4 Performance results: Signature space overhead 38

6.1 Summary of security properties 39

Contents

Abstract **i**

1 Introduction **1**
1.1 Background . . . 3
1.1.1 Syslog . . . 3
1.1.2 Cryptographic primitives . . . 3
1.1.3 Forward security . . . 4
1.1.4 Hash-chains . . . 4
1.1.5 Forward secure asymmetric signature schemes . . . 5
1.1.6 Trusted Platform Module . . . 5

2 Related Work **7**
2.1 Hash chains . . . 7
2.2 Forward Secure Sequential Aggregate (*FssAgg*) Signature Schemes 8
2.3 Syslog extensions . . . 9
2.4 Alternatives to Itkis-Reyzin's asymmetric FSS scheme . . . 9
2.5 Remote Code Execution Attestation & Dynamic Root of Trust . . 10
2.5.1 Cerium . . . 10
2.5.2 BIND . . . 10
2.5.3 Pioneer . . . 11
2.5.4 Flicker . . . 11

3 Signing Techniques to be Evaluated **14**
3.1 Traditional RSA-signing . . . 15
3.2 Traditional Hash-chains . . . 15
3.3 Itkis-Reyzin's asymmetric FSS scheme . . . 17
3.3.1 The algorithm . . . 17
3.4 RSA signing and tick-stamping with TPM . . . 20
3.5 Truncation detection technique . . . 22

4 Method **24**
4.1 Implementation . . . 24
4.2 Performance Testing . . . 24
4.3 Security evaluation . . . 25

5 Results **27**
5.1 Implementation details . . . 27
5.1.1 Logging framework architecture . . . 27
5.1.2 Traditional RSA-signing . . . 29
5.1.3 Traditional Hash-chains . . . 30
5.1.4 Itkis-Reyzin's asymmetric FSS scheme . . . 31
5.1.5 RSA signing and tick-stamping with TPM . . . 32
5.1.6 Truncation detection . . . 35
5.2 Performance . . . 36

6 Analysis **39**
6.1 Traditional RSA-signing . . . 40
6.1.1 Performance . . . 40
6.1.2 Security . . . 40
6.2 Traditional Hash-chains . . . 41
6.2.1 Performance . . . 41
6.2.2 Security . . . 41
6.3 Itkis-Reyzin's asymmetric FSS scheme . . . 42
6.3.1 Performance . . . 42
6.3.2 Security . . . 43
6.4 RSA signing and tick-stamping with TPM . . . 44
6.4.1 Performance . . . 44
6.4.2 Security . . . 44
6.5 Truncation detection . . . 47
6.5.1 Performance . . . 47
6.5.2 Security . . . 47
6.6 Man in the middle . . . 47

7 Conclusions and Future Work **49**
7.1 Conclusions . . . 49
7.2 Future Work . . . 50

References **52**

Chapter 1

Introduction

A log is a record of events which is being generated by an application or system. This can for example be security events such as firewall status, anti-virus events or user authentication events. In an investigation of a potential intrusion or system failure, these logs can contain vital information (ip addresses of remote connections at the time of the attack, for example). However, the volume and variety of logs generated by a computer system today makes managing them a very complex task. In order to ease the task of managing logs in a computer system, centralized log management systems has been developed with the purpose of collecting, analyzing and storing log data generated by log clients in a network setting. [20]

For logs to be a source of digital evidence, regardless of the setting, they must be deemed trustworthy. In order for log data to be admitted as evidence there are several legal requirements which has to be met, of course these requirements are highly dependent on situation and local legislations. An important step in order to deem log data as trustworthy is to validate its integrity, i.e. to make sure that it has not been tampered with in any way. [2]

Periods where a client is disconnected from the company log management system, e.g. a business trip, puts strain on the trustworthiness of the data being transmitted when the connection is reestablished; how can the integrity of log data be validated when collected by the central log server? Indeed the same question can be posed even when log records are being transmitted in real-time and this can be of importance when dealing with automated malware that has ability to modify log entries.

Scenario For the purpose of this thesis we assume the log client (software) to be running on a portable device (e.g. a laptop) and has been set up to be part of a corporate network with central log management. The user of the device may on occasion go on business trips, taking the device with her. During such periods the device may be disconnected form the central log management server for an arbitrary length of time and it is not inconceivable that during that time the device gets infected with malware or is hacked by a third party.

Once the attack has occurred and if the attacker is favoring stealth she does

not attempt to stop the log client from trying to reconnect with the central log management system. The attacker can however attempt to modify any entries locally buffered by the log client software before these entries are sent to the server when the connection is reestablished.

This scenario serves merely as a device to keep the reader informed on the motivation and scope of this thesis. The solutions presented in this thesis are not limited to the above scenario.

Scope/Research questions Given that the log client has been compromised as described by the scenario above, this thesis attempts to answer the questions below. These questions are posed from the perspective of both a forensic investigation and a central automatic anomaly detection system, or Intrusion Detection System (IDS).

1. What are the threats to the integrity of log entries generated prior to a compromise of a portable log client?

2. What existing and novel techniques can be used to verify the integrity of log entries generated prior to a compromise of a portable log client?

3. What advantages and disadvantages does the different techniques provide with regards to security properties and performance?

In the scenario described in this thesis the attacker cannot actually be prevented from gaining complete control of the stream of log entries. Thus this thesis focus on the ability to *detect* modification, insertion and/or deletion of log entries generated prior to the attack. This capability would also serve as a deterrent against such modifications and it would allow a central IDS-system to flag suspicions activity based on either verified suspicious entries or on verification failures on unsuspicious entries.

A discussion on the difficulty of detecting modifications to the stream of log entries generated *after* the attack can be found in chapter 6.

Of course, once the attack has occurred the attacker may instead simply delete any locally buffered log entries and prevent the log client software from running. This cannot be prevented but its detection is trivial.

Contribution In order to answer these questions we have implemented a prototype suite of logging software, consisting of client, server and verifier applications. In this prototype we have included three existing methods of providing integrity assurance of log entries, as well as one novel technique using the *Trusted Platform Module* (TPM, see section 1.1.6). This thesis provides theoretical overviews in addition to more detailed implementation, performance and security evaluations for each of the above techniques.

Recognizing the inability of the evaluated techniques to detect so called truncation[1] attacks we have developed a truncation-detection module, completely independent from each of the above mentioned techniques, also included in the prototype implementation.

Outline In section 1.1 we begin by briefly summarize the background of the work performed in this thesis. In chapter 2 we follow up by shortly describing work performed by others in the same field (that is, secure logging). Chapter 3 offers a theoretical overview of each signing technique studied in this thesis, where we also briefly describe our truncation detection module. In chapter 4 we present our methodology and in chapter 5 we present the implementation details and the result of the performance evaluation. We analyze our findings in chapter 6 and finally we summarize the thesis in chapter 7.

1.1 Background

1.1.1 Syslog

We decided to follow the syslog protocol [22, 14] in our implementation, due to it being the de-facto standard logging protocol used by many Unix and Linux based operating systems. By following this standard we hope to reduce the friction caused by migrating from one logging system to another.

The original syslog protocol was not designed with security or reliability in mind [22], it uses UDP and transmit all logs in plain text. RFC 5424 and RFC 5425 [14, 29] was published in 2009 and detailed the new syslog standard that uses more reliable (TCP) and secure (TLS) transport protocols. Several extensions to the syslog protocol has also been suggested [2] (see section 2.3).

1.1.2 Cryptographic primitives

In order to provide assurances of data integrity a cryptographic concept known as *digital signatures* can be used. A digital signature is a mathematical construct that can be used to prove that data has not been modified since signing and to authenticate its origins. This is possible since only a party knowing a secret key (also known as private key) may produce the signature. Any party knowing the corresponding public key may verify the signature. One commonly used signature scheme of this kind is based on the RSA encryption method [19]. This kind of scheme is the first method being implemented and evaluated in this thesis.

Another possibility is using a *Message Authentication Code* (MAC) [12] which also can be used to confirm integrity and authenticity, however, MAC's operate

[1]Removal of one or more log entries at the very end of a log, see section 4.3.

by using a *shared* secret key known to both parties. This means that anyone knowing this shared secret may alter a signature without it being detectable.

1.1.3 Forward security

Since an attacker can recover any secret key present on a compromised machine both digital signatures and MAC's can be forged undetectably. This means that any signed log entries locally stored on the machine may be undetectably modified by an attacker, even if the entry was signed before the attack occurred. To combat this the concept of *forward security* was introduced.

Forward security, also known as forward integrity, was first formally defined by Bellare & Yee [6] in 1997. The term is an application of *forward secrecy* to the field of digital signatures, forward secrecy is a property of key-agreement protocols first defined by Diffie *et al.* [11]:

> An authenticated key exchange protocol provides perfect forward secrecy if disclosure of long-term secret keying material does not compromise the secrecy of the exchanged keys from earlier runs. The property of perfect forward secrecy does not apply to authentication without key exchange.

I.e. the purpose of forward security is to mitigate damage caused by key exposure in a digital signature scheme, i.e. prevent the integrity of signatures signed prior to the key exposure from being compromised. In the field of secure logging, this would in practice prevent an intruder from forging log entries generated before root access to the machine was established, as an attempt of doing so would alert the administrators of his presence.

1.1.4 Hash-chains

To demonstrate their newly defined security property, Bellare & Yee presented a signature scheme which used *Message Authentication Codes* (MAC) in a new way [6]. Instead of generating all signatures using the same secret key, the key changes with every new *epoch*. An epoch is a predetermined amount of log entries or period of time for which each key is valid, where every new key is derived using a one-way hash function from the key of the previous epoch.

Whenever a new key is generated the key from the previous epoch must be securely erased from the system, this ensures that if the machine is compromised the intruder can only tamper with log entries generated during the current or future epochs without it being detected.

The initial key can be used to verify (and alter) all log entries, regardless of to which epoch it belongs, and should therefore be stored remotely. To detect altered or deleted log entries the verification process traverses the audit log from

the beginning; a missing sequence number or a mismatching MAC will indicate that the log has been tampered with. Note that this kind of solution cannot prevent an intruder from deleting or editing log entries, it can only detect it after the fact.

These principles are the basis of the seconds technique which has been implemented and evaluated in this thesis.

1.1.5 Forward secure asymmetric signature schemes

In 1999 Bellare & Miner [5] presented a forward secure public key signature scheme. Forward security is achieved by letting the secret key irreversibly evolve over time (the same concept as described in the previous section) together with a static public key which is able to verify signatures signed using any of the secret keys (following the principles of *digital signatures* presented above). The scheme has been proven forward secure in the *random oracle model*[2]. Improvements to this scheme has later been proposed which is said to increase the practicality of this scheme by significantly reducing the key-length required for signing [1].

In 2001 Itkis & Reyzin [18] presented a forward secure public key signature scheme based upon a signature scheme presented in 1988 by Guillou & Quisquater [16]. The security of this scheme is based on the *strong RSA assumption*[3] [4, 13] and provably secure in the random oracle model. This is the third scheme to be implemented and evaluated in this thesis, no prior available implementation of this technique is known to the authors.

1.1.6 Trusted Platform Module

The main concern with using traditional digital signatures is the lack of forward security. When the private key is compromised in an attack all signatures created prior to the attack are vulnerable to forging. We have already discussed two ways of solving this (hash-chains and forward secure asymmetric signature schemes), however, it occurred to us that the *Trusted Platform Module* might also be used to achieve the same goals. A solution using the TPM has been designed, imeplemented and evaluated by us in this thesis.

The flowing can be found on the homepage of the Trusted Computing Group (TCG) [15]:

> The TPM is a microcontroller that stores keys, passwords and digital certificates. It typically is affixed to the motherboard of a PC. It

[2]The random oracle model is a mathematical abstraction typically used in mathematical proofs of security when certain cryptographic hash functions cannot be proven to posses some mathematical properties required by the proof. The security of a system is then proven in the random oracle model as opposed to the standard model.

[3]The strong RSA assumption states that it is difficult, given n as the product of two unknown primes, to find $A \in \mathbb{Z}_n^*$ and $b > 1$ such that $A^b \equiv C \pmod{n}$ for any value $C \in \mathbb{Z}_n^*$.

> potentially can be used in any computing device that requires these functions. The nature of this silicon ensures that the information stored there is made more secure from external software attack and physical theft. Security processes, such as digital signature and key exchange, are protected through the secure TCG subsystem. Access to data and secrets in a platform could be denied if the boot sequence is not as expected. Critical applications and capabilities such as secure email, secure web access and local protection of data are thereby made much more secure. TPM capabilities also can be integrated into other components in a system.

A Trusted Platform Module is a passive hardware module whose specification is determined by the Trusted Computing Group (TCG). The TPM can be used as a trusted provider of cryptographic services and can make *attestations* of the current machine state.[8]

Chapter 2

Related Work

The works presented in this chapter are relevant to secure logging field in general and has largely influenced the work performed in this thesis, they do not however contain any knowledge required for understanding the work presented in this thesis.

We start this chapter by exploring further developed signing schemes based on the hash-chain principle, we then continue on to discuss so called FssAgg signature schemes and how they could be of use in our scenario. We also discuss some extensions to the syslog protocol and list some alternatives to the “Itkis-Reyzin's asymmetric FSS scheme” scheme. We also present some work done in the Remote Code Execution Attestation & Dynamic Root of Trust field, since those may potentially be of use for securing log clients.

2.1 Hash chains

Schneier & Kesley [34] further improved upon the hash-chain concept presented by Bellare & Yee, as well as creating a protocol to securely set up the initial authentication key with a remote trusted server. The scheme developed by Schneier & Kesley does not divide an audit log into epochs, rather each log entry contains an element in a hash chain. This means that once the hash-chain is proved to be intact, validating an arbitrary log entry also authenticates the integrity of all log entries prior to the one being validated. Another addition to the work of Bellare & Yee is the encryption of the log data using a symmetric algorithm. Each log entry is encrypted using a unique key which is derived from the authentication key together with viewing rights for that specific log entry, which also is an addition made by Schneier & Kesley. This allows different users with different rights to decrypt only the log entries they have the permission to read. The encrypted data is then used for creating the next hash in the hash-chain which in turn is used to generating the MAC, the reason for this is to make it possible to authenticate the integrity of the audit log even if you do not have the rights to decrypt all entries.

The security of this scheme, just as all hash-chain based schemes, hinges on the fact that the initial authentication key is kept secret, and that every time a

new authentication key is generated, the old one is irretrievably deleted. That is why Schneier & Kesley also created a protocol which sets up a log file with an initial secret which is then stored on a remote trusted server, which also allows a third semi-trusted party to verify an audit log file and if it has the appropriate permissions, also read it. However, this requires an online connection to function properly.

Chong *et al.* [10] describes a solution where the Schneier-Kesley scheme is used together with an external tamper-resistant hardware module in the role as the trusted server. The purpose of this is to eliminate the need for an online connection when creating new audit logs. However, the logs still need to be verified by a remote instance which means that some online requirements still exist.

J. E. Holt[17] describes an implementation of the Schneier-Kelsey protocol and also describing some performance and convenience features not previously considered. It adds on the Schneier-Kelsey protocol by using public key cryptography in order to make the verification process unaware of the secret root key[1].

R. Accorsi[3] has devised a scheme that provides a "digital black box". The author demonstrates that truncation attacks are possible against both the original Schneier-Kelsey protocol and the one described by Stathopoulos *et al.* R. Accorsi modifies the Schneier-Kelsey protocol and focuses on both storage and transmission phases. By using PKI[2] each log entry is signed before transmission to the central log storage server. This ensures the origin of the entries. The server then signs the hash chain links itself, providing an audit trail. The scheme provides for resistance against replay attacks and truncation detection on top of the integrity and audit trail assurance provided by the original protocol.

2.2 Forward Secure Sequential Aggregate (*FssAgg*) Signature Schemes

This type of signature schemes were first introduced by Ma & Tsudik [24] and provide a way to sign multiple entries in a forward secure way while aggregating previous signatures so that verification of a single signature verifies the entire log up to that point. This provides a space efficient solution on systems with where storage space and/or communication bandwidth is limited.

It also has the interesting security property of being able to detect truncation attacks since it stores a single aggregated signature apart from the log entry chain and each previous version of the signature must be securely overwritten. If a previous signature is recovered by an attacker then a truncation attack can be performed to that point. This also has the unfortunate side effect that if signature

[1] The key used to derive all other keys.

[2] Public Key Infrastructure.

verification fails then *no* log entries can be verified in the chain, there is no way to detect up to what point the chain remains valid, if an attack occurs. This means that the server would need to validate each signature before appending new log entries to its storage and thereby increasing its load and implementation complexity.

Due to the fact that there can only exist one signature for the entire log entry chain, the signature must be securely erased from all previous entries.

Several later papers [23, 37] have been published, detailing one or more alternative *FssAgg* schemes, however they all follow the same fundamental principles.

2.3 Syslog extensions

Several extensions to the syslog protocol exist, the ones most relevant to this thesis are presented below.

syslog-sign [7] is a theoretical (i.e there is no implementation known to the authors) log transmission protocol that improves upon the standard syslog protocol by adding a signature block to each log entry. This signature block is created by concatenating the hashes of the last three entries (including the current one). The result is then hashed again and this represents the signature block. Syslog-sign ensures the integrity of transmitted log entries. It also provides detection of deleted entries and replay resistance. Syslog-sign is a transmission phase only protocol, and does not provide for confidentiality of the transmitted data.

reliable syslog [30] implements reliable delivery, device authentication, log entry integrity and replay resistance.

2.4 Alternatives to Itkis-Reyzin's asymmetric FSS scheme

Itkis-Reyzin's scheme has been implemented and evaluated in this thesis, presented below are some alternative schemes based on the same principles.

KREUS ("Kozlov-Reyzin Efficient Update Signatures") [21] is a forward secure asymmetric signature scheme with a faster update algorithm than any earlier presented schemes, this allows for smaller intervals between key updates which in turn increases security. Even though the signing and verifying algorithms of this scheme are not as fast as some other similar schemes it is described as "reasonably efficient". As in the case of "Itkis-Reyzin's asymmetric FSS scheme", the security of this scheme relies on the *strong RSA assumption*.

MMM (Malkin, Micciancio, Miner) [26] is a *generic* forward secure signature scheme, meaning that it can be based on any underlying signature scheme. MMM is the first forward secure signature scheme which does not require the number of time periods a secret key is valid for to be pre-defined, instead there exist only a

theoretical limit to how many times the key update algorithm can be called. This upper limit is an exponential of a security parameter and, for practical values, cannot feasibly be reached. Also, the security of this scheme has been proven in the *standard model*[3], i.e. without relying on the random oracle model.

2.5 Remote Code Execution Attestation & Dynamic Root of Trust

In the scenario described by this paper the following concepts are relevant in that they could potentially be used to verify that log entries has indeed been signed by the correct piece of application logic (PAL) (i.e. "code") and that its data is correct. Since the log data ultimately comes from a range of sources, attestation and verification of each source is impractical. Therefore the works below will be briefly examined to see if they could potentially be used as bases for alternative implementations of our "RSA signing and tick-stamping with TPM" technique.

2.5.1 Cerium

The authors of *Cerium*[9] proposed a new trusted computing architecture using tamper resistant CPU and a µ-kernel[4]. The µ-kernel uses separate address spaces for each running program, together with other memory protection techniques this ensures that each program can only access their own data. Each running program is cryptographically authenticated and copy protected by the CPU/µ-kernel each time the program code and data is stored in the untrusted DRAM (that is, each time the CPU-cache lines are evicted the CPU traps to the µ-kernel).[9]

The CPU also signs secure certificates that identifies the CPU, its manufacturer, the BIOS, boot loader, µ-kernel, running program and any data that the program wants signed. These certificates can then be used for verification of the program, its environment and whether or not its output can be trusted.[9]

Of course this require special hardware that is not readily available and makes use of a special µ-kernel. Since this solution would not be available on existing Unix/Windows systems our interest in Cerium is purely academic in nature.

2.5.2 BIND

BIND is a "Fine grained Attestation Service for Secure Distributed Systems" [36]. Its operation depends on a Secure Kernel (SK) present in the system. The security of the SK is based on its small size and a static root of trust (such as provided by UEFI's Secure Boot[31] or using Trusted Boot as described by [8].

[3]In the standard model an adversary is limited only by the amount of time and the amount of computational power available to him.

[4]A very small kernel.

By using the SK, BIND can by means of a TPM verify the hash signature of a PAL (piece of code) immediately before executing it. The SK is also responsible for setting up a protected environment around the PAL before its execution and to verify its input data. When the PAL terminates the SK signs any output data generated by the PAL in such a way that the data can be tied to the code that produced it. [36]

The lack of SK in existing systems such as Linux, Windows or OSX makes this approach unfeasible for the scenario proposed in this thesis.

2.5.3 Pioneer

In contrast to previously mentioned solutions to remotely verifiable code execution Pioneer [35] does not rely on specific hardware and can thus be used on legacy systems. Instead Pioneer relies on a distributed protocol that measures the client code by means of a hash and nonce and by arguing that the execution time will be longer for any attempt to forge the code signature. The execution time is then recorded by the remote host and if it exceeds a certain threshold value the client will no longer be trusted. [35]

This requires knowledge of the client hardware where the “real” execution time of the verification function is known and that this time does not change in any significant way (such as over-clocking the CPU). It also relies on the fact that the verification function is indeed the most optimal implementation for that particular machine. [35]

Since Pioneer solution requires a perpetually online verification server it is not a usable solution in the scenario of a disconnected client device that this thesis presents.

2.5.4 Flicker

Flicker [28] enables isolated execution of code with exclusive access to sealed data. This is the most practical alternative solution to date and it offer the developer a framework that bootstraps the system into a state of dynamic root of trust and executes a PAL that can be considered a minimal *Trusted Computing Base* (TCB). It can do this by using the `SKINIT` processor instruction on processors that has support for AMD’s *Secure Virtual Machine extensions* (SVM) or the `GETSEC[SENTER]` instruction on processors with support for Intel’s *Trusted eXecution Technology* (TXT). [28]

These instructions perform a so called *late launch* of a Virtual Machine Monitor (VMM) or Security Kernel (SK) at an arbitrary time after boot, with full protection against software based attacks. The instructions mentioned above takes as argument an address to a *Secure Loader Block* (SLB) in memory that is to be run. The processor uses hardware protections to protect the SLB from software attacks; it disables all interrupts, it disables direct memory access (DMA)

to the physical memory pages of the SLB and it even disables both hardware and software debugging access. Then it enters a flat 32-bit protected mode and jumps to the entry point of the SLB. [28]

The TPM includes a number of *Performance Counter Registers* (PCRs) that can be used for attesting the hardware and software state of the machine. Each of these registers can be *extended* with a new measurement. A measurement can in reality be anything that can be represented as a SHA-1 value.

In order to attest that the SLB has been properly executed it commands the TPM to zero PCR registers 17-23 and then measure the SLB (by means of hash) and extend that into PCR 17. There is no way for software to reset PCR 17 without executing another *late launch* instruction. This means that the value of PCR 17 can be used in attestations in order to confirm that the proper SLB has indeed been loaded. [28]

Flicker is built as a framework that instead of providing a SLB that launches a VMM or SK it instead runs a small PAL that is run until its completion and then after some cleanup Flicker restores the previously running operating system providing the rest of the application with a memory address containing the output of the PAL. [28]

Because the *late launch* instructions are privileged instructions there must be built-in support in the operating system in order to facilitate the required functionality to a user space application. In Linux this is done as a kernel module that exposes some *sysfs* entries. On the Windows platform it has been implemented as a kernel space driver. [27, 28]

The downside to this approach is that it freezes the currently running operating system and programs for the duration of the PAL runtime. This means that the PAL must be extremely short-lived for it not to have an adverse effect on the system, or being a nuisance to the user. Unfortunately the PAL-runtimes tested by the McCune *et al.* froze the system between 15 milliseconds and over one second. There is also the practical issues of it being (in it current state) an extremely unstable experimental solution. [28]

Potential alternative scheme based on Flicker - I

One way to leverage Flicker is to to design a PAL (Piece of Application Logic) that takes input (a log entry), inserts a timestamp and signature into the appropriate field and outputs the signed entry. The first time a signature is to be generated a different PAL will be loaded that generates a private key used for signing. That key will be sealed in the TPM so that it can only be unsealed by the PAL used for signing. At the same time a custom log entry will be generated signaling a new *epoch* and that a new key is to be used. The entry will also contain the public part of the key that will be used for verifying the signatures of all later entries.

The decision to trust the signatures should be based on whether the start of *epoch* occurred during a trusted state of the machine or not. *Epoch* is intended

to last the lifetime of the machine and as such should begin when the machine is installed, no other start of *epochs* should occur for that particular machine.

The main argument against the this technique is the fact that the performance evaluation in [28] estimates a runtime for the PAL of ∼1 second *for each log entry* during which the OS and all other applications are suspended from responding to input and performing any work. In the above cited [28] performance measurement the run-times was divided into different parts. The actual *late launch* instructions that was tested in that paper on that particular machine took ∼15 ms to perform if the necessary *unseal* operation was excluded form the measurement.

Potential alternative scheme based on Flicker - II

A solution that does not require the use of TPM seal/unseal for each log entry, to combat the drawback of the solution explained above, is described below.

When Flicker is launched the *late launch* instruction calculates the hash of the SLB (including the PAL) and extends PCR17 with that value. The SLB executes the PAL and when the PAL exits the SLB (Flicker) extends PCR17 with hashes of inputs to and outputs from the PAL. A constant well-known value is then also extended to PCR17 in order to prevent access to data sealed by the TPM to that particular PAL.

The main idea here is that the PAL should only be responsible of generating a timestamp (output) for each log entry (input). Since the signature of both inputs and outputs to the PAL is extended to PCR17 the TPM quote mechanism should render it virtually impossible for any software to forge an attestation of the combined signatures of the PAL, its input (log entry) and its output (timestamp).

The main advantage of this approach is that the quote operation is performed *after* the OS has been resumed which, in theory, should give a mere ∼15 ms of operating system freeze time for each log entry, without decreasing the security of the signatures (since PCR17 cannot be reset without using the *late launch* instructions).

These 2 schemes were devised by the authors of this thesis but they were not included for evaluation for the simple reason that they do not provide any more log entry security than the "RSA signing and tick-stamping with TPM" technique. The argument being that the only data that could not be forged are signature and timestamps itself, the log entry that was signed could still be provided (or held back) by an attacker. This renders the technique vulnerable to exactly the same attacks as the "RSA signing and tick-stamping with TPM" technique.

Chapter 3

Signing Techniques to be Evaluated

Each technique presented in this thesis will be implemented by us in a new extensible logging framework, consisting of a client, a server and a standalone verification application. We have identified the following fundamentally different techniques to sign log entries on a disconnected client:

- Traditional RSA-signing (existing technique).
- Traditional Hash-chains (existing technique).
- Itkis-Reyzin's asymmetric FSS scheme (no known publicly available implementation).
- RSA signing and tick-stamping with TPM (novel technique).

The following sections will present each of the above techniques in more detail, but from a purely conceptual and theoretical point of view.

We recognize the inability of the above signing techniques to detect so called *truncation attacks* (defined as A4 in section 4.3). We have therefore developed a truncation detection scheme which can be used together with any of the above signing techniques. This scheme is presented in section 3.5.

Security concepts For each of the above signing techniques we have presented a table specifying which security properties the current technique is imbued with, each of these are explained below.

Origin & content integrity assures the verifier that the signed data can only come from a party knowing the secret key and also that the data has not been modified in any way after it was signed.

Stream integrity assures the verifier that the order of individually signed data packets has not been modified in any way. This property also ensures that no data packet has been inserted or deleted from the stream.

Forward security assures the verifier that even if the current secret key has been revealed, data signed prior to the key exposure can not be undetectably modified.

Secure time/tick-stamp assures the verifier of the time of signing in a way that cannot be undetectably be altered by an attacker.

Verification by public key allows verification of signatures without the secret key being known by the verifier.

3.1 Traditional RSA-signing

Origin & content integrity	Stream integrity	Forward security	Secure time/tick-stamp	Verification by public key
yes	no	no	no	yes

Table 3.1: RSA signing is the simplest technique to implement, but it has the least to offer in terms of log entry security

By signing each log entry with a private key the integrity and origin of these entries can be verified, and as the private key is not needed for verification it is never permitted to leave the machine. However, once the private key has been recovered by an attacker she may use it to forge and delete any entry that has not yet been transmitted to the server. Indeed, if the attacker gains access to the storage server then all entries may be modified in any way without risk of being detected.

This technique does not provide *forward security* nor can it be used to verify the *stream integrity* of each log entry since the same private key is used for signing each log entry, and once the key is revealed any part of the log chain may be forged. The reason for including this technique in the thesis is to provide a baseline for comparison, both in terms of security and performance.

3.2 Traditional Hash-chains

Origin & content integrity	Stream integrity	Forward security	Secure time/tick-stamp	Verification by public key
yes	yes	yes	no	no

Table 3.2: Hash-chains are a simple and efficient way of generating forward secure signatures, though due to its symmetric nature it is vulnerable to root key exposure in a way which other solutions are not.

Hash chains provides content integrity by generating a Message Authentication Code (MAC) for each log entry. A MAC is a cryptographic construction used to verify the integrity of data, as well as its origin. The scheme presented here

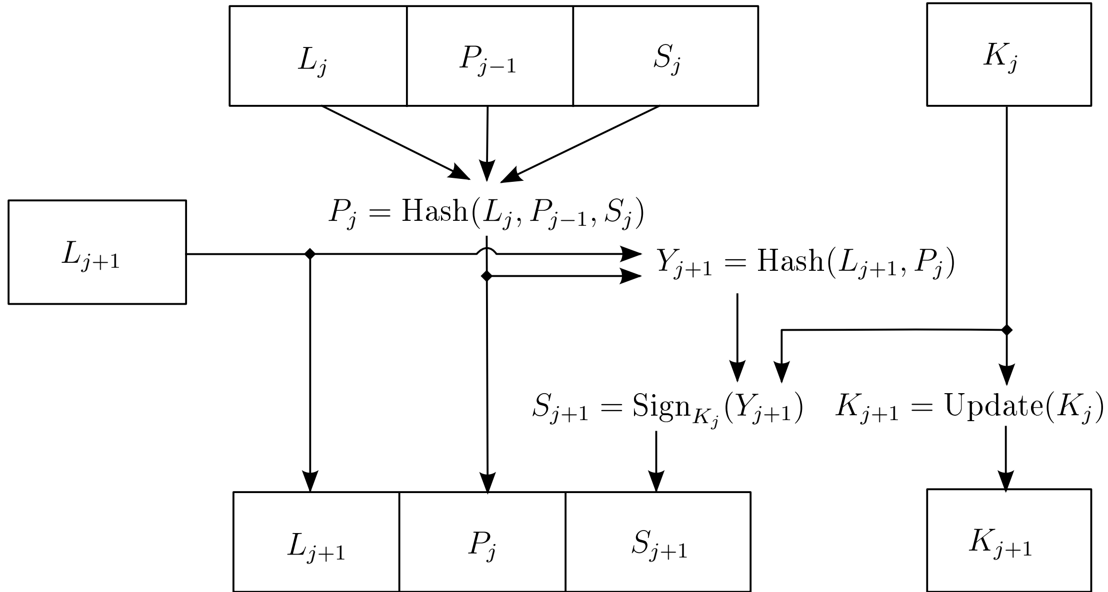

Figure 3.1: Schematic overview of the our generic signing process employed by "Traditional Hash-chains" and "Itkis-Reyzin's asymmetric FSS scheme". L_{j+1} is the actual content of the next log entry, P_j is the hash of the previous entry, S_{j+1} is the outputted signature of the entry and K_j is the current secret key.

makes use of HMAC (specified in FIPS 198-1 [12]) which is a version of MAC that uses cryptographic hash functions together with a secret key to generate a MAC.

The HMAC method takes as input the arbitrary length data to be signed as well as the key with which it is to be signed. In our version of this scheme we sign the hash of the current log entry which also contains the hash of the previously signed entry (see figure 3.1), the reason for this is to provide a way to confirm stream integrity i.e. detect if any log entries has been deleted or inserted into the log stream.

A key is only ever used once to sign a log entry before it is evolved, this evolution is simply done by hashing the current key and using the result as the new key. It is this mechanism that ensures forward integrity of the scheme, since hashing is a one-way function it is infeasible to recover previous keys which would make it possible to forge earlier entries. During the initial setup a root key is randomly generated and securely synchronized to a trusted server, the security of this entire scheme hinges on the fact that this key is secret and it is therefore of utmost importance that this key is securely removed from the client machine.

3.3 Itkis-Reyzin's asymmetric FSS scheme

Origin & content integrity	Stream integrity	Forward security	Secure time/tick-stamp	Verification by public key
yes	yes	yes	no	yes

Table 3.3: "Itkis-Reyzin's asymmetric FSS scheme" improves upon traditional asymmetric signature schemes by providing stream integrity and forward security.

The first (to our knowledge) asymmetric signature scheme with an evolving secret key and a static public key was proposed by Bellare & Miner in 1999 [5]. Several similar schemes has been proposed since then and the one we have decided to implement has been designed by Itkis & Reyzin [18], this due to its supposed efficient signing and verifying.

This scheme generates a signature by performing a series of mathematical operations which requires the signer to have access to the secret key of the current epoch. The data we sign in our implementation consists of the hashes of both the current and the previous log entries, also in this case to provide a way to detect missing entries. As the key evolves with time and previous keys can not be recovered afterwards this scheme also provides the forward security property, while at the same time the public key used for verifying signatures remains static which eliminates the need to synchronize any secret key with the remote server.

For the process of the actual log entry chain generation we follow the same scheme as presented in "Traditional Hash-chains" (see figure 3.1), although the key update does not always occur for each log entry. When the key-update function actually runs is further discussed in section 5.1.4.

3.3.1 The algorithm

Key generation The key generation algorithm takes as input three security parameters k, l and T and returns the public key PK and the initial secret key SK_1. k and l are key sizes which are used when generating primes. T is used to determine the number of times the private key can be updated (evolved), e.g. if $T = 365$ and the key evolves once per day this public-private key pair will be valid for exactly one year. The algorithm is summarized in pseudo code in figure 3.2. [18]

Key update The key update algorithm takes the current secret key SK_j, where $j < T$, and outputs the next secret key SK_{j+1}. The algorithm is summarized in pseudo code in figure 3.3. [18]

```
function KEY GENERATE(k,l,T)
    Generate random (k/2 − 1)-bit primes q_1 and q_2, s.t. p_i = 2q_i + 1 is prime
    n ← p_1 p_2
    t_1 ← random integer from Z*_n
    for i = 1, 2, ..., T do
        e_i ← prime such that 2^l (1 + (i − 1)/T) ≤ e_i < 2^l (1 + i/T)
    end for
    f_2 ← e_2 · ... · e_T (mod φ(n)) where φ(n) = 4 q_1 q_2
    s_1 ← t_1^{f_2} (mod n)
    v ← (s_1^{e_1})^{−1} (mod n)
    t_2 ← t_1^{e_1} (mod n).
    PK ← (n, v, T)
    SK_1 ← (1, T, n, s_1, t_2, e_1)
    return PK, SK_1
end function
```

Figure 3.2: "Itkis-Reyzin's asymmetric FSS scheme" key generation algorithm

```
function KEY UPDATE
    Let SK_j = (j, T, n, s_j, t_{j+1}, e_j)
    if j = T then
        return                    ▷ The key has reached the end of its lifetime.
    end if
    Regenerate e_{j+1}, ..., e_T.
    f_{j+2} ← e_{j+2} · ... · e_T.
    s_{j+1} ← t_{j+1}^{f_{j+2}} (mod n).
    t_{j+2} ← t_{j+1}^{e_{j+1}} (mod n).
    return SK_{j+1} ← (j + 1, T, n, s_{j+1}, t_{j+2}, e_{j+1})
end function
```

Figure 3.3: "Itkis-Reyzin's asymmetric FSS scheme" key update algorithm

Signing The signing algorithm takes the current secret key for a time period $j \leq T$ and a message M as inputs and produces a signature S. The algorithm is summarized in pseudo code in figure 3.4. [18]

```
function SIGN(M)
    SK_j = (j, T, n, s_j, t_{j+1}, e_j).
    r ← a random integer from Z*_n.
    y ← r^{e_j} (mod n).
    σ ← H(j, e_j, y, M) where H is a hash function.
    z ← r s^σ (mod n).
    return S ← (z, σ, j, e_j).
end function
```

Figure 3.4: "Itkis-Reyzin's asymmetric FSS scheme" signing algorithm

Verification The verification algorithm takes the public key, a message M and a signature S as inputs and verifies that S is a valid signature for M. The algorithm is summarized in pseudo code in figure 3.5. [18]

```
function VERIFY(M, S)
    Let PK = (n, v, T).
    Let S = (z, σ, j, e_j).
    if e < 2^l or e ≥ 2^l(1 + j/T) or e is even then
        return False                          ▷ The signature is invalid
    end if
    if z ≡ 0 (mod n) then
        return False                          ▷ The signature is invalid
    end if
    Let y' ← z^{e_j} v^σ (mod n).
    if σ = H(j, e_j, y', M) then
        return True                           ▷ The signature is valid
    else
        return False                          ▷ The signature is invalid
    end if
end function
```

Figure 3.5: "Itkis-Reyzin's asymmetric FSS scheme" verification algorithm

The success of the verification relies on the verifier being able to recompute y' and thereby σ to the same value as produced during the signature generation. This is possible due to the following.

By definition we have that

$$s_i^{e_i} \equiv v^{-1} \pmod{n} \text{ for } 1 \leq i \leq T \tag{3.1}$$

and therefore we get that

$$\begin{aligned} y' &\equiv z^{e_j} v^{\sigma} \\ &\equiv (r s_j^{\sigma})^{e_j} v^{\sigma} \\ &\equiv r^{e_j} \cdot (s_j^{e_j})^{\sigma} \cdot v^{\sigma} \\ &\equiv r^{e_j} \cdot (v^{-1})^{\sigma} \cdot v^{\sigma} \\ &\equiv r^{e_j} \cdot v^{-\sigma} \cdot v^{\sigma} \\ &\equiv r^{e_j} \\ &\equiv y \pmod{n}. \end{aligned} \tag{3.2}$$

This means that to be able to forge a signature from an earlier time period an attacker is required to acquire an earlier s_i given s_j for a period $i < j < T$. This is, according to the *strong RSA assumption*, infeasible modulo n.

3.4 RSA signing and tick-stamping with TPM

Origin and content integrity	Stream integrity	Forward security	Secure time/tick-stamp	Verification by public key
yes	yes	yes	yes	yes

Table 3.4: "RSA signing and tick-stamping with TPM" is the, in theory, most secure technique that this thesis evaluates. Although it does require specialized hardware to be of use.

In order to test the possibility of utilizing the hardware modules included in many existing systems, a novel log signing protocol was developed. This technique makes use of the *Trusted Platform Module* function `Tspi_Hash_TickStampBlob` in order to be securely provided with a signed *tick-stamp*. The tick-stamp also includes a hash of any binary data blob provided by the user, thus ensuring that the data existed some time *prior* to, and has not been modified since, the wall clock time implicated by the tick-stamp.

The reason for not utilizing previous work such as Flicker (see section 2.5.4) was due to the practical limitations and dependencies imposed. By instead implementing a purely TPM based solution a higher level of compatibility can be ensured. It will be shown in this thesis that there will actually be no loss of security properties by giving up the DRTM[1] and remote code execution attestation features of those works.

[1] Dynamic Root of Trust.

This log entry signing technique provides signing capability through the use of a non-migratable private RSA key generated and secured by the TPM. The TPM ensures that this key never leaves the secure internal storage of the TPM, unless encrypted by its *Storage Root Key* (SRK). This property ensures that the log entries must have been signed by the same TPM that created the key and therefore allowing verification of both *origin-* and *content integrity*. It also provides *verification by public key* due to the use of the private key for signing. The verification can therefore be processed on a different machine knowing only the the public key. This also means that there is no need to keep track of a secret verification keys, that may potentially leak and be used to forge log entries.

The tick-stamp from the TPM contains the following data: *data nonce*, *data hash*, *current ticks*, *tick rate*, *tick nonce* and *signature*. The meaning and use of data nonce is to provide security against replay attacks; by ensuring that each request to the `Tspi_Hash_TickStampBlob` function uses a unique value in this field the recipient may verify that the resulting signature has not used in a duplicate log entry. The data hash field is used to verify that certain data has indeed existed before the call to `Tspi_Hash_TickStampBlob`, this is ensured by the one-way property of the hash function.

The current ticks, tick rate and tick nonce fields are used to give an indication of how much time has passed since last boot of the machine. The current ticks field is implemented as a monotonic counter that is incremented each clock cycle (the frequency of which is provided as the tick rate field). The TPM ensures that the field can only be zeroed by a reset or cold boot of the machine, on each a random nonce is written to the tick nonce field in order to prevent replay attacks. It is therefore possible to distinguish between different boots of the machine. The signature field, of course, contains the signature of all the above fields (and of the log entry itself by way of the data hash field).

The `Tspi_Hash_TickStampBlob` provides *forward security* and *stream integrity* since no entries may be altered undetected after it has been signed. The detection is based on the tick-stamp; since any new signature includes the tick-stamp it is straight forward to detect if any entry has been signed out of order or during a wall clock time that does not correspond with the content of an entry (such as the system time provided by the timestamp field).

Of course, *after* a machine has been compromised the attacker may can send (or do not send) whatever log entries to the signer she wishes and the verification process has no way of detecting this. It should be noted however that no matter the privilege of the attacker on the client machine she can never forge *when* these log entries were signed.

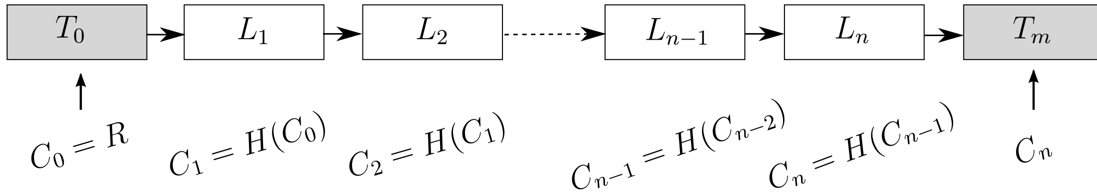

Figure 3.6: During the initial setup the counter (C_0) is initialized with a random value (R) which is inserted into a special start entry (T_0) and then transmitted to the server. For each encountered log entry (L) the counter is incremented by hashing its current value. Each time the local log entry buffer is transmitted to the server it is appended with a special close entry containing the current value of the counter (T_m, where m is the number of end-tags sent).

3.5 Truncation detection technique

A truncation attack on the above techniques does not break the stream integrity of the log entry chain, since each log entry only contains enough information to validate the existence of prior log entries.

To combat this we use a one-way counter (implemented with a one-way hash value). This counter is initialized with a random value (this value is transmitted to the server via a special initialization message) and for each log entry the value is updated simply by hashing the current value. The previous value of the counter is securely erased from the system every time the counter is updated.

The key idea behind this approach is that the log client generates a special *close entry* each time the local log entry buffer is uploaded to the server. The close entry simply contains the current counter value which the server can use to re-calculate the number of entries it was supposed to receive and this number can then be compared to the actual number of entries it has received. The server can do this calculation since it knows the secret random value used to initialize the counter. See figure 3.6.

If an attacker gains access to the machine at a time where there exists n number of signed log entries, there is no way for her to remove any entries and generate the special close entry expected by the server. This is due to the fact that there is no way for her to recover any of the previous counter values $C_{n-1}, \ldots, C_0$.

We stress to the reader that the only thing this scheme ensures is that at least n number of log entries has been encountered, this means that she can of course replace any log entries with forged entries, though this would be detected given that they are also signed by a forward secure scheme. There is also the possibility that the attacker prevents the transmission of these close entries, however, this would of course be noticed by the server alerting the verifier to a probable truncation attack. Once the attacker has gained access to the client and knowledge of the counter value C_n any truncation attacks on future entries L_k where $n < k$

are naturally undetectable.

Chapter 4
Method

To improve the ability to securely verify the contents and timings of log entries on devices, without a persistent network connection, the techniques presented in chapter 3 has been implemented and investigated in terms of performance, security properties and operational requirements.

4.1 Implementation

In order to facilitate this investigation a new cross platform logging framework has been developed that includes the signing methods and the truncation detection technique. Also a server and a verification application is developed for testing of the entire log chain pipeline.

4.2 Performance Testing

In order to test the performance of each signing technique the following metrics has been measured/evaluated in an identical environment:

- **Maximum throughput** – The maximum number of log entries that the current signing technique can handle for any length of time. The length of each log entry randomly varies between 50 and 200 characters to simulate a realistic environment.
- **Key generation** – The time it takes for the "setup" phase to complete. If applicable the latency of key *re*-generation is also measured.
- **Configuration parameters** – All configurable parameters has been evaluated for its performance impact. Values will be chosen in such a way that comparison of each technique will be as straightforward as possible.
- **Disk space overhead** – The amount of additional space per log entry required by each signing technique, i.e. the number of characters added.

The absolute values of these metrics are not what is important but the relative differences between the techniques are, it is therefore important that the tests are running on the same machine in the same environment.

We assume that verification is done on demand in an investigation and not necessarily performed as an automatic process. As such the verification process will not be measured since it should have no impact on suitability of the signing technique in question.

4.3 Security evaluation

The security of each technique has been analyzed by identifying what attacks each log signing technique can detect[1] during verification. We assume that the attacker gains complete control of the machine software (i.e. kernel level privileges) at time $t = T_A$.

If we also assume that a point i in the log entry chain has been created at $t = T_i^{\text{create}}$, signed at $t = T_i^{\text{sign}}$ and transmitted to the server at $t = T_i^{\text{trans}}$ then the attacks on the log entry chain at i can be categorized according to its relationship with the time of the attack (T_A). Of course the relationship $T_i^{\text{create}} < T_i^{\text{sign}} < T_i^{\text{trans}}$ always applies. The lifetime of a log entry can be summarized as in equation (4.1).

$$\begin{aligned} &\text{creation } (T_i^{\text{create}}) \to \text{signing } (T_i^{\text{sign}}) \to \text{local storage} \to \\ &\qquad \to \text{transmission } (T_i^{\text{trans}}) \to \text{server storage} \to \text{verification} \end{aligned} \tag{4.1}$$

The attacks that can be performed on the log entry chain can be categorized based on the current relationship of T_A and T_i. An explanation of each category follows the list of attacks below.

- If $T_A < T_i^{\text{sign}}$ then these attacks are available to the attacker:

  A1. Append forged log entry chain.[2]

  A2. Timestamp control.[3]

- If $T_i^{\text{sign}} < T_A < T_i^{\text{trans}}$ then these attacks are available to the attacker:

  A3. Deletion[4]

  A4. Truncation[5]

[1] Since the attacker has complete control of the machine nothing can actually be *prevented*.
[2] The act of appending to the end of the existing log entry chain (excluding attack A2).
[3] The act of forging the timestamp information in new log entries.
[4] The act of deleting one or more existing entries in the middle of the chain.
[5] The act of deleting one or more existing entries at the end of the log entry chain.

A5. Modification[6]

A6. Insertion[7]

- If $T_i^{\text{trans}} < T_A$ then these attacks are available to the attacker:

  A7. Deletion on server

  A8. Truncation on server

  A9. Modification on server

  A10. Insertion on server

  A11. Appending on server

The attacks A1 and A2 can be performed on the client on all log entries not yet signed or created when the attack has occurs, this is simply a consequence of that the attacker has full control of the machine software and may (at the very least) decide what information are fed into the signing module. A2 is a special case of A1 that may or may not be possible to detect regardless of the privileges of the attacker (see "RSA signing and tick-stamping with TPM").

A3-A6 are attacks where the attacker attempts the modify log entries that were already signed when the attack occurred. It is primarily these type of attacks the techniques presented in this thesis is trying to detect.

A7-A11 are the same attacks as A3-A6 but with the distinction that these occur on the server instead. This distinction is important when considering what information (e.g. secret keys) also need to be stored on the server (or someplace else, but potentially available to a sophisticated attacker).

[6] The act of modifying the content of an existing entry.

[7] The act of inserting a new fabricated entry in the middle of the log entry chain.

Chapter 5

Results

In this chapter we present the implementation details of: our logging framework, each log entry signing method and the truncation detection technique. We also present the results of the performance measurements in this chapter.

5.1 Implementation details

5.1.1 Logging framework architecture

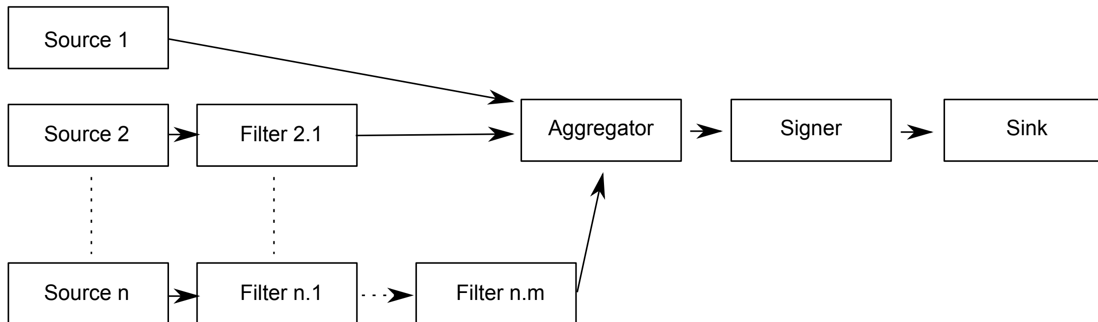

Figure 5.1: The log entry chain is processed in a pipeline manner, where each box in the figure represents its own POSIX-thread. The arrows indicate how log entries are transferred from one step to the next.

Overview

The framework was developed for Unix and Microsoft Windows using `C++` and `boost.build.v2` with the following core dependencies: `boost`, `crypto++`, `OpenSSL` and `sqlite3`.

The architecture is build around a pipeline approach (see figure 5.1) where each step is represented by its own thread. The pipeline steps are: `LogSource`, `LogFilter`, `LogAggregator`, `LogSigner` and `LogSink` where each step feeds the next in the pipeline.

There can be multiple `LogSource`'s and each can have any number of `LogFilter`'s in a chain, but every chain ends with the same `LogAggregator` which aggregates all entries into a single queue for the `LogSigner` and `LogSink`. The `LogSink` is responsible both for the temporary local storage on the client and the transmission to the server when a connection is available. For a list of currently available implementations of each pipeline step see table 5.1.

The implementation is open sourced under a MIT-license at `https://bitbucket.org/anma-exam-2014/prototype-implementation`.

LogSources	LogFilters	LogAggregators	LogSigners	LogSinks
DummySource	RegexFilter	SimpleTimestamper	DummySigner	DummySink
FileSource		AntiTruncation-Timestamper	RsaSigner	SyslogTlsSink
WindowsSource			HashChainSigner	
			ItkisReyzinSigner	
			TpmRsaSigner	

Table 5.1: List of currently implemented pipeline steps in our prototype log client.

The Syslog protocol

In order to facilitate easier integration with existing technologies we have followed the syslog protocol as defined in RFC 5424 [14] and its TLS protocol transport mapping as defined in RFC 5425 [29].

The signature format is naturally dependent on the signing technique in question, but in common for all is that the signature is contained in the "Structured Data" of the syslog protocol. This structure is a list of named arrays of key-value pairs, that can be used for including arbitrary data in structured manner. It follows the following generic format:

```
[CUSTOM@32473.1.2 KEY1="VALUE" KEY2="VALUE"][OTHER@32473.1.2
OTHER_KEY2="VALUE" OTHER_KEY2="VALUE"]
```

Where "`CUSTOM@32473.1.2`" is the id of the structured data defined as `name@<private enterprise number>` (32473.1.2 in this case is just an example), anyone can define an id in the above format provided they are part of an organization that has a "SMI Network Management Private Enterprise Code" as maintained by IANA (the *Internet Assigned Numbers Authority*).

The usage of id's without `@` signs is also regulated by IANA, excerpt from RFC 5424:

> Names that do not contain an at-sign ("@", ABNF %d64) are reserved to be assigned by IETF Review as described in BCP26 [RFC5226]. Currently, these are the names defined in Section 7. Names of this format are only valid if they are first registered with the IANA. [...]

Persistent Storage Service

All pipeline steps have access to a "Persistent Storage Service" which enables each step to save important information that must be preserved between restarts of the software. This service has been implemented using `sqlite3`. The local buffering of log entries are handled by the sink separately. In the case of `SyslogTlsSink` this has also been implemented using `sqlite3`.

Special considerations

RFC 5425 specifies that no application level acknowledgments are to be sent back from the server to the client upon receiving data. The reason for this is most likely a result of considering performance and/or implementation complexity [33]. As a consequence however it leaves the client in the dark about whether the log entries sent has actually been received or not.

The client and server applications has therefore been extended with the option to send and receive an application level acknowledgment. The acknowledgment is simply a number specifying how many log entries has been received and successfully stored by the server. The connection is immediately terminated if an error occurs on the server side, this allows the client to restart the connection and try again. Together with SQLite transactions this ensures that the log file will always be consistent.

Upon receiving the reply the client can then delete these entries from its local storage and keep sending more entries to the server. If it is the servers response that is lost and not the entries themselves, the client will resend the related entries. Because of this a duplication detection algorithm has also been implemented on the server.

By doing this the client can ensure that no entries are lost due to terminated network connections or if the client itself is terminated abruptly. To ensure maximum compatibility this feature can be configured on or off.

5.1.2 Traditional RSA-signing

Signing

When instantiated for the first time `RsaSigner` (as it is called in the implementation) generates by default a 2048-bit RSA key.

When the new key is generated a special log entry is created and put in the queue to be send to the server with the following fields in its "Structured Data", where `"..."` is integers represented as strings:

```
[RSASIGNATURE@41717 PUB_N="..." PUB_E="..."]
```

`RsaSigner` generates signatures by following the RSASSA-PSS (*RSA Signature Scheme with Appendix-Probabilistic Signature Scheme*) algorithm (according to PKCS #1 v2.1 / RFC 3447 [19]) used with the SHA-256 hash algorithm.

The structured data of each signed log entry has the following format, where `"..."` is a base64 encoded string:

```
[RSASIGNATURE@41717 SIGNATURE="..."]
```

Each log entry is signed by first generating its string representation with the `SIGNATURE` set to an empty string in its "Structured Data". The resulting signature over the log entry is base64 encoded and written to the `SIGNATURE` field.

Verification

The verification is equally straight forward as the signing. The value of the `SIGNATURE` field in "Structured Data" of the log entry is verified against the complete string representation of the log entry itself, with the value of the `SIGNATURE` field replaced with an empty string (`"..."` in the example above).

5.1.3 Traditional Hash-chains

Signing

The `HashChainSigner` initially generates a random 256-bit root key. When the new key is generated a special log entry is created and put in the queue and sent to the server. This log entry has the follownig format in its "structured data", where `"..."` is the hex-encoded value of the root key:

```
[HASHCHAINSIGNATURE@41717  KEY_HASH="..."]
```

If a key already exist from a previous run of the program, the key is instead loaded from persistent storage. Once the key initialization is complete the `HashChainSigner` is ready to sign log entries. Before the log entry is signed an empty placeholder value of the signature is inserted. If a log entry has been signed previously the hash of that entry is also inserted into the current log entry before signing. A signature is produced by running the hash of the log entry through a HMAC (specified in FIPS 198-1 [12]) together with the current version of the key. The signature is then inserted into its empty placeholder in the structured data section of the log entry. The `HashChainSigner` uses the following format in the "structured data" of each log entry, where "..." is data encoded as a hexadecimal string:

```
[HASHCHAINSIGNATURE@41717  SIGNATURE="..." PREVIOUS="..."]
```

Each key is only ever used to produce one signature and must therefore be updated before another signature will be signed, this is done by simply hashing the current key and using the resulting hash as the new one.

Verification

The `SIGNATURE` field contains the signature which is to be verified, this is achieved by simply computing a new signature from the supplied entry (its hash and the supplied hash of the previous entry). If the generated signature matches the one supplied in the `SIGNATURE` field it is considered valid, otherwise not.

In addition to simply verifying the validity of a signature, another goal of the verification process is to detect entries which are missing. This is achieved by comparing the value supplied in the `PREVIOUS` field with the hash computed from the previous entry. If they do not match, one or more log entries are missing and possibly deleted. In a situation where entries are missing the current key will most likely not be correct for that particular entry. To try and recover from this, the affected entry will be be verified with the next 1000 keys (or until a successful verification) in an attempt to compute the number of missing entries.

5.1.4 Itkis-Reyzin's asymmetric FSS scheme

Signing

The `ItkisReyzinSigner` (as it is called in our framework implementation) generates a key from the supplied parameters `k`[1], `l`[2] and `T`[3] (These were explained in more detail in section 3.3). The public key is then sent (together with some key setup parameters) to the remote server as an ordinary log entry, with the following structure:

```
[ITKISREYZINSIGNATURE@41717 PUB_N="..." PUB_V="..." PUB_T="..."
PUB_L="..."]
```

where "..." are string representations of the corresponding integer values (base 10). In the case where a key already exists, it is instead loaded from persistent storage.

The lifetime of a key is specified as T periods where, depending on the configured mode, one period can either be a period of time (`key-evolve-time`), a number of signatures (`key-evolve-sign`) or both, whichever comes first. At the end of each period the secret key is updated which is the start of a new period.

Log entries are hashed and signed together with the hash of the previously signed entry in order for the verifier to detect missing entries.

```
[ITKISREYZINSIGNATURE@41717 SIG_Z="..." SIG_SIGMA="..."
SIG_E="..." SIG_J="..." PREVIOUS="..."]
```

where all values are string representations of integers except `PREVIOUS` which is a base64-string of the previous entry's hash.

[1] default value `k=2048`
[2] default value `l=128`
[3] default value `T=1000`

Verification

The fields `SIG_Z`, `SIG_SIGMA`, `SIG_E` and `SIG_J` contains the different elements of the signature and `PREVIOUS` contains the hash of the previous entry, these are extracted and used to verify the signature. The previous hash is included in order to be able to detect entries that are missing. Since the public key is static it can verify all signature which have been signed by a secret key associated with the public key.

Special consideration

In Itkis & Reyzin's paper they described an optimization which would speed up the key update algorithm. They reduced the size of the primes generated significantly which should reduce the execution time, but when implementing this optimization the execution time increased by a factor of 3. Our theory is that the algorithm used by `Crypto++` is much less efficient when generating smaller primes opposed to larger ones, though due to time limitations we have not had the time to investigate this further and therefore this optimization has been not been included in our implementation of the algorithm.

5.1.5 RSA signing and tick-stamping with TPM

Overview

This technique introduces the additional dependencies on both the software library `TrouSerS` and (more limiting) the TPM hardware module. The `TpmRsaSigner` (as it is called in the framework implementation) generates a 2048 bit RSA key if it has not already been created on the client. If such a key has already been created it reads it from the persistent storage of the `TrouSerS` library (not the "Persistent Storage Service" provided by the logging framework) installed on the machine.

As mentioned in section 3.4 the key is encrypted by the TPM SRK (Storage Root Key) and cannot be used until loaded by the TPM which decrypts the key and ensures that the key never leaves the internal storage of the TPM in its unencrypted form.

When a new key is generated a special log entry is created with the following keys in "Structured Data"

```
[TPMRSASIGNATURE@41717 VERSION="1" BATCH="0"
SIGN_KEY_EXPONENT="..." SIGN_KEY_MODULUS="..."]
```

Where "..." is data encoded as a base64 string. `BATCH="0"` identifies this entry as the first enty outputted since the key was generated. Each signature operation increments this number by 1.

In order to increase throughput of the `TpmRsaSigner` the concept of batches was introduced. This means that the signer can concatenate all log entries in the queue and sign these in one operation. The first entry in every batch has the following fields in its “Structured Data”:

```
[TPMRSASIGNATURE@41717 BATCH="X" INDEX="0"
BATCH_SIZE="Y" SIG_INFO="..." SIG="..."]
```

This identifies the log entry to be part of signing batch X and that the verifier should expect there to be $Y - 1$ other entries in the batch. The next entry in the same batch has the following fields in its “Structured Data”:

```
[TPMRSASIGNATURE@41717 BATCH="X" INDEX="Z"]
```

Where Z is the position of the log entry in the batch. The actual data that is signed is the concatenation of all log entries in the syslog format with all “Structured Data” fields set to their expected values *except* `SIG_INFO` and `SIG` which are set to a zero length string. The tick-stamp information received from `Tspi_Hash_TickStampBlob` follows the format of `TPM_SIGN_INFO` (see figure 5.2) where `data` is a concatenation of the hash of log entry and a `TPM_CURRENT_TICKS` structure. The signature format follows that of `PKCS#1 v1.5` with the `SHA-1` hash function.

Signing

The signing is done by first creating a `TSS_VALIDATION` structure (see figure 5.3), calling it `v`, and letting `v.rgbExternalData` point to a random anti-replay nonce (and "`v.ulExternalDataLength = sizeof(TPM_NONCE)`").

The data to be signed is both the hash of the log entry data (or a batch of several entries) and the tick-stamp information provided by the TPM.

The command to generate the actual signature is the `Tspi_Hash_TickStampBlob` function with a pointer to the signing key, hash object (for the log entry data) and the `TSS_VALIDATION` structure as inputs.

The output of this function is pointed to by the `v.rgbData` and the `v.rgbValidationData` pointers respectively. Note that the contents of `v.rgbExternalData` is actually copied into the (`TPM_SIGN_INFO`) structure pointed to by `v.rgbData`.

It is the data pointed to by `v.rgbData` and `v.rgbValidationData` that is encoded into the `SIG_INFO` and `SIG` fields of the “Structured Data” of the syslog protocol.

```
1  typedef struct tdTPM_SIGN_INFO {
2      TPM_STRUCTURE_TAG  tag;
3      BYTE  fixed[4];
4      TPM_NONCE  replay; //anti replay nonce
5      UINT32  dataLen;
6      BYTE*  data;  //HASH(log entry) || TPM_CURRENT_TICKS
7  } TPM_SIGN_INFO;
8  typedef struct tdTPM_CURRENT_TICKS {
9      TPM_STRUCTURE_TAG  tag;
10     UINT64  currentTicks; //The number of ticks since last boot
11     UINT16  tickRate; //The number of ticks per ms
12     TPM_NONCE  tickNonce; //Random value for telling boots apart
13 }TPM_CURRENT_TICKS;
```

Figure 5.2: Excerpts from the TCG TPM specification 1.2 [25] where the `TPM_SIGN_INFO` and `TPM_CURRENT_TICKS` structures are defined. Comments in green are added by the authors to indicate usage specific to this thesis.

```
1  typedef struct tdTSS_VALIDATION
2  {
3      TSS_VERSION  versionInfo;
4      UINT32       ulExternalDataLength;
5      BYTE*        rgbExternalData; //anti replay nonce
6      UINT32       ulDataLength;
7      BYTE*        rgbData; //points to a TPM_SIGN_INFO struct
8      UINT32       ulValidationDataLength;
9      BYTE*        rgbValidationData; //The signature
10 } TSS_VALIDATION;
```

Figure 5.3: Excerpt from the TCG TPM specification 1.2 [25] where the `TSS_VALIDATION` structure is defined. Comments in green are added by the authors to indicate usage specific to this thesis.

Verfication

The field `SIG` contains the signature of the data in `SIG_INFO` field, and as such is validated separately from the integrity of the log entry itself. However since the hash of the log entry is included in the `SIG_INFO` field the log entry may be verified by simply comparing the log entry hash with a newly calculated hash over the same data, as described above.

The verifier must also consider actual values of the other fields contained in the `SIG_INFO` structure, such as `currentTicks`, `tickRate` and `tickNonce` and compare those to the values of the same fields in the preceding log entries. Also the `BATCH` and `INDEX` fields must be checked to ensure that no entries are duplicated or lost.

Special Considerations

The TCG specification[25] states that `"UINT16 tickRate"` (see figure 5.2) is defined as:

> The number of microseconds per tick. The maximum resolution of the TPM tick counter is thus 1 microsecond. The minimum resolution SHOULD be 1 millisecond.

However while testing the verification the inverse (i.e. "*The number of ticks per microsecond*") appeared to be the only usage that resulted in remotely correct figures on our test machine.

The accuracy of the `tickRate` field has not been thoroughly tested, also the `tickRate` field seems to be constant (100 on the test machine). In our limited testing there has been great variations on the reported `tickRate` and the calculated `tickRate`. We speculate that the majority of this discrepancy is due to the fact that the timestamp in the log entry itself does not report the time of signing but rather the time of the creation of the log entry. This discrepancy is something the verifier must be aware of.

5.1.6 Truncation detection

Overview

During the initial setup a random 256 bit counter value is generated as C_0 (see figure 3.6). When C_0 has been generated a special initiation entry is sent to the server with the following format, where `"..."` is the base 64 representation of C_0.

```
[ANTITRUNCATION@41717 INIT="..."]
```

Every time a log entry passes through the `AntiTruncationTimestamper` (as it is called in our implementation) the counter is updated, this simply by applying

one round of SHA-256 hashing to the current value. A special end-tag entry is sent to the server when all of the following conditions are met:

- Log sink has sent a notification that the local buffer is empty.
- Log sink has sent a notification that a connection to the server is currently available.
- The above state has been unchanged for at least x number of milliseconds, where x is a configurable parameter.
- An end-tag is not the last sent entry.

The counter value is not affected by the above mentioned end-tag entries. End-tag entries has the following format, where `"..."` is the base 64 representation of C_n.

```
[ANTITRUNCATION@41717 HASH="..."]
```

Verification

Verification is performed by recalculating the C_n values from the initial value C_0 for as many log entries that has been received, if this recalculated C value matches the value provided in the last end tag then no truncation has been detected. If the values does not match the verifier will attempt to calculate the amount of missing entries by comparing up a fixed number of counter increments.

There are also the possibility of missing end-tags which simply means that truncation can not be disproved.

Space optimization

By not signing end-tag entries they can be removed by the server without breaking any eventual signed log entry chains. This is done in our prototype implementation before any new end-tag entries are inserted, thus ensuring that only the last end-tag remains on the server.

This optimization could be important in environments where a persistent network connection to the log server is prevalent. On such systems, depending on the configuration, every other entry could potentially be an anti-truncation end-tag. This could require up to twice the amount of disk space, compared to a log file without any anti-truncation end-tags.

5.2 Performance

The results of the performance tests with regards to signing throughput has been summarized in table 5.2. As an attempt of measuring the overhead induced by

	Parameters	Throughput (entries/second)
No signing (overhead)	–	16438
Traditional RSA-signing	–	72.9
Traditional Hash-chains	Including key update	2767
Itkis-Reyzin's asymmetric FSS scheme	Excluding key update	100.7
RSA signing and tick-stamping with TPM	$N = 100$, One entry per batch	2.8
RSA signing and tick-stamping with TPM	$N = 10000$, All entries in one batch	10383

Table 5.2: The performance results of all signing methods, they were benchmarked by measuring the time it took to sign 1000 entries (N) with random content lengths (between 50 and 200 characters), unless specified otherwise by the *Parameters* column. The content lengths were randomized in order to simulate realistic behavior, and the results above are calculated averages of ten consecutive runs to compensate for this randomness

the simple iteration of all entries and generating their frame[4] the signing method "No signing (overhead)" was also measured (by implementing a `DummySigner`). This will serve as a baseline for comparison.

The results of the performance tests with regards to key setup and key updates has been summarized in table 5.3.

Some methods are included multiple times in order to differentiate between different input parameters impact on performance. The results displayed in the tables are calculated averages over ten consecutive runs.

It is important to note that the absolute values are not what is interesting but rather the relative values in relation to one another, a more in-depth discussion of the results can be found in chapter 6.

The test machine that was used (due to it being the first available machine with a *Trusted Platform Module*) was a *Lenovo Thinkpad Helix* with 8192 MB of RAM and an Intel Core i7-3667U processor running at 2 GHz with Ubuntu 13.10 32-bit.

The disk space requirements for each signing technique is summarized in table 5.4.

No performance tests with regards to verification efficiency has been performed in this thesis due to it not being considered relevant in this scenario.

[4]The string representation of the log entry as specified by RFC 5424 [14].

	Parameters	Key setup (s)	Key update (s)
Traditional RSA-signing	$k = 2048$	0.313	–
Traditional Hash-chains	$k = 256$	5.31×10^{-3}	3.01×10^{-5}
Itkis-Reyzin's asymmetric FSS scheme	$k = 2048$, $l = 128$, $T = 100$	20.7	0.587
	$k = 2048$, $l = 256$, $T = 100$	14.2	1.31
	$k = 2048$, $l = 128$, $T = 1000$	17.5	3.89
	$k = 2048$, $l = 256$, $T = 1000$	23.3	11.2
RSA signing and tick-stamping with TPM	$k = 2048$	1.46	–

Table 5.3: The performance results of key setup and key update (if applicable). Input options to key setup/update are specified in the *Parameters* column. The results are above are calculated averages of ten consecutive runs.

	Initialization entry size (characters)	Overhead per log entry (characters)
Traditional RSA-signing	754	378
Traditional Hash-chains	178	179
Itkis-Reyzin's asymmetric FSS scheme	≈ 1379	≈ 860
RSA signing and tick-stamping with TPM	434	533

Table 5.4: Disk space usage overhead for each log signing technique. Note that some of these figures might vary greatly depending on the parameters and inherent randomness of each technique.

Chapter 6

Analysis

This chapter provides an analysis of each log signing technique, focusing both on the results of the performance evaluation presented in section 5.2 and presenting a security evaluation. The security evaluation takes the attacks presented in section 4.3 and tries to determine which of those can be detected from a theoretical point of view (i.e. we disregard the possibility of implementation errors).

A summary of detectable attacks against each technique is provided for the reader in table 6.1.

Attacks defined in section 4.3: → / Techniques defined in chapter 3: ↓	A1: Appending	A2: Timestamp	A3: Deletion	A4: Truncation	A5: Modification	A6: Insertion	A7: Server Deletion	A8: Server Truncation	A9: Server Modification	A10: Server Insertion	A11: Server Appending
Traditional RSA-signing	No	No	No	No	No	No	Yes	No	Yes	Yes	Yes
Traditional Hash-chains	No	No	Yes	No	Yes	Yes	No	No	No	No	No
Itkis-Reyzin's asymmetric FSS scheme	No	No	Yes	No	Yes	Yes	Yes	No	Yes	Yes	Yes
RSA signing and tick-stamping with TPM	No	Yes	Yes	No	Yes	Yes	Yes	No	Yes	Yes	Yes

Table 6.1: A summary of the security properties of each log signing technique. "Yes" means that the technique can detect the attack, "No" means it cannot detect the attack. Note that there may be some requirements that must be fullfilled the the attack to be detectable, in that case these are detailed in their respective section.

6.1 Traditional RSA-signing

6.1.1 Performance

The performance result presented in table 5.2 indicates as expected that normal RSA signing is significantly slower than Hash-chains and performs similarly to the "Itkis-Reyzin's asymmetric FSS scheme". This is almost without question due to them both being based on the same mathematic principles for encryption (or in this case signing). It does however, require less space during storage than any of the other asymmetric signature scheme presented in this thesis as we see in table 5.4, and only "Traditional Hash-chains" require less disk space.

6.1.2 Security

As already mentioned in section 3.1 this technique does not offer much in terms of log entry security other than what is provided by the asymmetric nature of the technique (*Origin & content integrity* and *Verification by public key*).

As will be shown below all attacks detailed in section 4.3 that takes place on the *client* are possible on this technique and it should *not* be used to sign log entries with the intention of it being used as a way to verify logs that has been in possible contact with a malicious software/attacker. It can however be used to detect certain kinds of attacks performed on the server if the requirements detailed below are fulfilled.

Attacks A1-A2

As described in section 4.3 these attacks are performed on log entries created *after* the attacker has gained complete control over the client machine. Since the private key used to sign is located on the machine the attacker can use it directly to sign any log entry she chooses.

There is also the possibility to instead simply feed the existing signing module with any log entry the attacker chooses instead of implementing her own signing procedure that mimics the same protocol put in place by the client software.

Attacks A3-A6

These attacks are performed by the attacker on log entries *already* signed (but not yet sent to the server) by the signing module when the attacker gains complete access to the client machine.

Since all entries are signed by the same private key and the key resides on the client the attacker can easily forge the log entry chain from any arbitrary point in the local chain and forward.

Of course, if the attacker gains access to the client *and* the server it can then perform the above attacks on the server as if it also were a client.

Attacks A7-A11

These attacks are performed by the attacker but instead of getting complete control of the client she gains read and write access to the log storage on the server.

If the public key used for verification is *also* stored on a secondary secure location then that can later be used to verify that the attacker has not changed the private key used for signing, thus rendering all attacks against the log entry chain on the server detectable (since the attacker does not have access to the private key). The exception to this is the A8 attack (truncation) since there is no way to detect missing entries at the end of the chain, with this signing technique.

Of course, if the attacker gains write access to all copies of the public key used for verification *and* has read and write access to the entire log storage on the server the entire chain may be replaced by a forged one by using a different public-private key pair for signing.

The assumption made that the attack A7 can be detected depends on the existence of some kind of sequence number specifying the index of the current entry in the log chain.

6.2 Traditional Hash-chains

6.2.1 Performance

As expected the "Traditional Hash-chains" produced the highest throughput of all schemes (excluding schemes with batching enabled[1]). This is due to the relatively lightweight computations required compared to for example RSA signing. Also, as expected it is the most space efficient of all the signing techniques that we have covered.

6.2.2 Security

As previously mentioned in 3.2, "Traditional Hash-chains" provides good client security (*Origin & content integrity*, *Stream integrity* and *Forward security*) with great performance. Though, because it lacks *Verification by public key*, it is vulnerable to key recovery on the server side which could, if an attacker gained control of the server machine, compromise the integrity of all signed log entries.

Attacks A1-A2

"Traditional Hash-chains" offers no security for log entries which has not yet been signed, this means that if an attacker were to gain complete control of the client

[1] Such as "RSA signing and tick-stamping with TPM".

machine she could append any log entry she wished to the current log chain without it being detected.

Attacks A3-A6

Since earlier keys cannot be recovered given the current one, “Traditional Hash-chains” does protect already signed entries from being modified, deleted from or inserted into the log entry chain without it being detected. Attempting any of these attacks will cause the chain to break which will be detected during the verification. Truncation (A4) however, cannot be detected through this method since the chain is never technically broken.

Attacks A7-A11

“Traditional Hash-chains” is not an asymmetric scheme and therefore requires the server to store the initial secret key. This means that, even with no access to the client, if an attacker gains control of the server she will be able to recover the root key. Once the root key has been compromised it will be possible to undetectably perform any attack in this category.

6.3 Itkis-Reyzin’s asymmetric FSS scheme

6.3.1 Performance

“Itkis-Reyzin’s asymmetric FSS scheme” produces almost the same (slightly higher) throughput that “Traditional RSA-signing” does, this was expected due to their similar nature. However, this is without considering the time consumed by the key update algorithm.

Depending on the way the scheme has been configured, the key update could take several seconds to perform which would significantly reduce the throughput of this method. Also, during the key generation phase “Traditional RSA-signing” is significantly faster than “Itkis-Reyzin’s asymmetric FSS scheme”, but since keys might have to be regenerated during run-time it might lower the throughput significantly.

As a result “Itkis-Reyzin’s asymmetric FSS scheme” might not be an option when it comes to solutions where performance is critical. However, since we have not been able to implement the optimizations proposed in the original paper (see section 3.3 and [18]) improvements in this area are possible.

The value of T has great impact on the execution time of both the key generation and key update algorithm, as it decides how many sub-keys the secret key will contain. Because of this, T has to be chosen with great care since a too low value will trigger the key regeneration algorithm to often and a too high

value will cause the key update algorithm to take a long time, severely impacting performance in a negative way.

As described in 5.1.4, `ItkisReyzinSigner` has three different modes of operation which controls how often the key update algorithm is being run, `key-evolve-time` or `key-evolve-sign`. The purpose of this is to provide some flexibility when it comes to key updates. `key-evolve-time` will in most cases offer the best performance due to the fact that it will only require key updates at specific times.

The k parameter controls the bit size of the modulus and should therefore be same length as a RSA key which is considered adequately secure (the default is 2048). l controls the size of the primes which generated at the setup and then regenerated at every key update and therefore have a great impact on performance. The size of l is recommended to be similar to the length of symmetric cipher key which is considered to be secure (default is 128). [18]

The storage space efficiency of "Itkis-Reyzin's asymmetric FSS scheme" is not very high as evidenced by table 5.4. This is mainly due to a combination of its asymmetric nature and the relatively high amount of parameters that is part of the signature.

6.3.2 Security

"Itkis-Reyzin's asymmetric FSS scheme" is, with exception to "RSA signing and tick-stamping with TPM", the algorithm which provides the best security, as described in 3.3, and only lacks *Secure time/tick-stamp*.

Note that, as we mentioned in section 3.3, this scheme is only forward secure between epochs. This means that within each epoch this scheme offers the same security properties as "Traditional RSA-signing". As a consequence the epochs should be kept as small as possible in order to decrease the number of log entries susceptible to forgery.

Attacks A1-A2

The moment an attacker gains complete control of the host machine she will be able to recover the current secret key and with this be able to successfully sign any future log entry she pleases. However, signatures from earlier time periods cannot be forged, due to the fact that it is not feasible to recover previous versions of the secret key given only the current secret key.

Attacks A3-A6

Since signatures are created in a chain like manner (each signature contains the hash of the previously signed entry), all attacks in this category except A4 (truncation) will be detectable during verification. A4 affects only the end of the chain

which means that the chain is technically never broken, this attack is comparable to a "Denial of Service" attack.

Attacks A7-A11

Assuming the attacker does not have access to the log client or any relevant secrets it may contain, an attacker will not be able to perform any attacks except A8 (again truncation) on the server due to the same reasons as described in the previous section.

6.4 RSA signing and tick-stamping with TPM

6.4.1 Performance

It comes as no surprise that signing with the TPM is the slowest technique presented in this thesis since it has the additional overhead of reading tick-stamp, hashing and signing on a separate hardware module. Also the current implementations of the TPM are notoriously slow[2].

Due to the the poor performance the concept of batches cannot be called anything but a significant improvement to such a degree that it even beats the "Traditional Hash-chains" technique.

This technique does use more disk space than "Traditional RSA-signing" due to the presence of the `SIG_INFO` field containing the, among other things, `currentTicks` and `tickRate` fields. The concept of batches again improves this due to the fact that only the first entry in the batch contains the `SIG` and `SIG_INFO` fields, the actual improvement in practice has however, not been investigated in further detail.

6.4.2 Security

We believe this technique to be the most secure of the techniques presented in this thesis and is therefore recommended to be further investigated in terms of both security of the implementation and increased ease of setup.

Attacks A1-A2

As described in section 4.3 these attacks are performed on log entries created *after* the attacker has gained complete control over the client machine. No signing method can be used to detect attack A1 since it is always possible for the attacker to feed the unmodified signing module whatever she likes.

On the other hand the TPM has the unique property of giving us an unforgeable tick-stamp indicating *when* the given log entry was signed. The tick stamp

[2]According to the TrouSerS FAQ `http://trousers.sourceforge.net/faq.html#3.3`.

is considered unforgeable because it is internal to the TPM itself and its value is signed by the signing key created during setup. Since the TPM makes sure the private key is never allowed to leave the TPM unless encrypted by its SRK (Storage Root Key, which is also internal to the TPM) the tick-stamp can not be signed or forged by any other entity. In other words the relative timings between log entries can be verified while the actual contents of the log entry (including the timestamp field of the syslog protocol) cannot be verified, on entries signed after an attack has occurred.

Attacks A3-A6

These attacks are performed by the attacker on log entries *already* signed (but not yet sent to the server) by the signing module when the attack occurred on the client machine.

Since any modification of the log entry chain means that the affected log entries must be signed again to avoid detection, the tick-stamp field will immediately indicate entries signed out of order in the chain and any such attack would be detected during verification.

It is although possible to pick a point in the chain and replace *all* log entries following that point with newly signed entries in order to hide the actual log chain modification, that would however put a time gap into the chain from the point of first originally signed entry to the time the signing of the forged chain occurred.

Note that it is not possible (as with all techniques described in this thesis) to detect the A4 attack (truncation) since it does not break the actual log entry chain. The attack is akin to a denial of service attack. The existence of such an attack may of course be indicated by the mere fact that no new entries actually arrive at the server.

Attacks A7-A11

Due to the asymmetric nature of the technique any attack (again except A8, truncation) performed on the server would be detectable since the attacker does not have access to the private signing key.

It is important that the attacker does not gain write access to all copies of the public key used for verification, since such access could be used to replace the signature of the *entire* log entry chain with one created by the attacker. It is therefore our recommendation that the public key is copied to (several) other machines or external media for safekeeping and used for verification at a later date, and since it is the public key it does not matter if it falls into the wrong hands as long as not *every* copy of it falls into the *same* hands.

Attacks on SHA-1

TPM version 1.2 and below unfortunately uses the SHA-1 hash algorithm in its signing process. SHA-1's collision resistance property is broken [38] and its use is discouraged [32]. The attacks on the collision property of SHA-1 means that it is possible in "less than brute-force" calculations to find another message M_2 that calculates to the same hash value H as the original message M_1.

By using this vulnerability it may be possible for an attacker to modify a log entry in such a way that the signature is still valid for the log entry. It should be noted however, that the modified message M_2 must, in practice, include some random looking data in order for it to actually hash to the same H. This should, at least in theory, mean that any such modifications could be detectable by a human investigator.

It may however be worse than that; If the attacker modifies the data in the `SIG_INFO` structure instead it may be possible for the attacker to instead of *appending* random data it actually *modifies* the content of the `nonce` field in such a way that the hash values collides. If the attacker do this then she may also modify the actual log entry and simply put the new hash of the modified log entry into the `SIG_INFO` structure.

The goal of newer implementations of this signing technique is, naturally, to move away from SHA-1 and instead use the much more secure SHA-2 family of hashes. Those are however, only available on the new TPM 2.0 specification and it will therefore limit the compatibility of this technique. In fact, the reason we did not implement a TPM 2.0 based solution at this date was due to lack of required hardware.

The inacurracy of the `tickRate` field

As a consequence of the inaccuracy of the `tickRate` field (first described in section 5.1.5), the `currentTicks` field can only very roughly be used to determine how long the machine has been running. It can however still be used as an indicator of if the timestamp in the log entry itself can be trusted (by comparing with previous entries). If the error rate between the `tickRate` and the timestamp of the log entry itself goes above or below certain threshold values then that would indicate that the timestamps or the system clock may have been manipulated/forged.

It must be noted by the verifier that the timestamp does not report the time of signing, but rather the time when the log entry passes the `LogAggregator` step of our log client prototype. Therefore in situations where the signer has a high load and cannot keep up the discrepancy between the tick rate and the timestamp could vary greatly. This means that one have to be careful when deciphering the meaning of the `currentTicks` and `tickRate` fields.

6.5 Truncation detection

6.5.1 Performance

We have not felt it necessary to perform any performance measurements on this module due to the relatively lightweight computations required to update the counter value.

6.5.2 Security

Using this technique a verifier can be assured that the anti-truncation module has received the same amount of log entries as indicated by the counter value, given that all of the following assumptions are met.

- An end-tag entry has actually been received by the server.
- No past counter values can be recovered, it is therefore important that:
  - All end-tag entries has been sent directly to the server without delay.
  - Previous counter values are securely overwritten in system memory and persistent storage.
- The one-way property of the currently used hash function holds true.

In the event that an end-tag entry has not been received by the server, it *may* be possible to extract the current counter value in a forensic analysis of the log client machine. If that is the case, that value could still be used to prove that no truncation attack has occurred on any log entry prior the index indicated by the counter value.

In the case of an attacker performing a truncation attack, she could naturally prevent the client from sending its latest end-tag entry. This would however, be suspicious since the server expects such entries to arrive after the configured idle timeout.

6.6 Man in the middle

A man in the middle attack against the log client and log server could in theory be performed. Such an attack would need to take place during the entire lifetime of the log entry chain since it needs to sign *all* signatures with its own secret key.

Asymmetric signature schemes The attacker would need to replace all transmitted copies of the verification public key with its own and somehow make sure that other copies of the key has not been transfered by another channel.

Hash-chains The attacker simply replaces the secret root key with its own, but it also needs to make sure that the key has not also (or instead) been transfered off the client by another channel.

Mitigation The man in the middle attack can be hampered by the TLS protocol and "certificate pinning" on both client and server. This has been done in the logging framework presented in this thesis, and it should therefore not be susceptible to such attacks, unless of course it gains access to the client or server, in which case the attack can be considered to be rendered moot.

Chapter 7

Conclusions and Future Work

7.1 Conclusions

In this thesis we have evaluated multiple different techniques of signing log entries on client machines. We have assumed the existence of a log storage server but assumed no persistent connection. We have provided for each signing technique performance comparisons and security evaluations.

Our contributions can be summarized as follows:

- Side-by-side performance comparison of existing and novel client-side log signing techniques.
- Side-by-side security comparison of existing and novel client-side log signing techniques.
- First known (to our knowledge) publicly available implementation of a technique using "Itkis-Reyzin's asymmetric FSS scheme".
- Presentation of a novel client-side log signing technique using the TPM hardware module.
- Development of a module used for ensuring that truncation attacks has not occurred on log clients.
- Development of an extensible open source log signing framework, including client, server and verification applications.

We have shown that the most secure signing technique ("RSA signing and tick-stamping with TPM") can potentially and under the right circumstances be used even after a machine has been compromised. The argument being that the `currentTicks` and `tickRate` fields cannot be forged and these may or may not be useful data in to an investigator, even if the data source of the log entry content itself cannot be verified.

We have also shown the validity of using asymmetric techniques such as "Itkis-Reyzin's asymmetric FSS scheme" on machines without a *Trusted Platform Module*. While it does not provide the secure tick-stamps of the TPM it does very

well to protect log entry data in the forward secure manner of its design. It also provides the advantage over "Traditional Hash-chains" in that the server and verifier never ever needs to learn the private signing key in order to verify the log entry chain. This, naturally, simplifies the key management greatly and removes many of its inherent security risks.

We have also shown that slow asymmetric techniques can be implemented in such a way (using batches) that performance should not be an issue even on the most hard pressed and low powered systems. There is little to no negative impact on security using the batch concept except that verification must be done over the entire batch instead of a per-entry basis.

Based on the results of this thesis we would recommend a move away from hash chain based log signing techniques and instead move to asymmetric techniques, due to the simplified key management those methods entail. However, the extra storage space requirements for these techniques can not always be ignored, and for that reason there may be some practical situations where a hash-chain based technique would still be the most appropriate.

7.2 Future Work

Due to our limited time we have not had the time to investigate more than one asymmetric forward secure signature scheme, therefore other forward secure asymmetric signature schemes should be investigated as possible alternatives to "Itkis-Reyzin's asymmetric FSS scheme", section 2.4 contains a more detailed discussion on this matter.

The "Itkis-Reyzin's asymmetric FSS scheme" technique must of course be more thoroughly vetted by the security community, both its theory and its implementation before it can be accepted for general usage.

Also the "RSA signing and tick-stamping with TPM"-technique in its current state uses TPM version 1.2 which in turn uses, as has been mentioned, the SHA-1 hash algorithm. This algorithms collision resistance property has been broken and as such any TPM based solution must be upgraded to the TPM 2.0 version, which uses more secure hashing algorithms, before it can be accepted for wide spread usage.

Some measure of protection against rouge clients should be implemented on the server side. Rouge clients could in theory attempt to perform a denial of service against the log storage server by flooding it with fake log messages. The server implementation should also be vetted against vulnerabilities that could be exploitable through a rouge client.

In order to increase the performance of the logging prototype we would also like to apply the batch concept to the other signing techniques presented in this thesis. The most performance gain is of course to be had with slower asymmetric algorithms rather than faster hash-chain based ones. Since the concatenation of

several entries takes both time and memory it is not entirely certain that hash-chain based signing techniques would show any measurable gain at all, this is something that would be very interesting to investigate at a future date.

References

[1] Michel Abdalla and Leonid Reyzin. A new forward-secure digital signature scheme. In *Advances in Cryptology—ASIACRYPT 2000*, pages 116–129. Springer, 2000.

[2] Rafael Accorsi. Log data as digital evidence: What secure logging protocols have to offer? In *Computer Software and Applications Conference, 2009. COMPSAC'09. 33rd Annual IEEE International*, volume 2, pages 398–403. IEEE, 2009.

[3] Rafael Accorsi. A secure log architecture to support remote auditing. *Mathematical and Computer Modelling*, 2012.

[4] Niko Barić and Birgit Pfitzmann. Collision-free accumulators and fail-stop signature schemes without trees. In *Advances in Cryptology—EUROCRYPT'97*, pages 480–494. Springer, 1997.

[5] Mihir Bellare and Sara K Miner. A forward-secure digital signature scheme. In *Advances in Cryptology—Crypto'99*, pages 431–448. Springer, 1999.

[6] Mihir Bellare and Bennet Yee. Forward integrity for secure audit logs. Technical report, Citeseer, 1997.

[7] Jon Callas, Alex Clemm, and John Kelsey. Rfc 5848: Signed syslog messages. 2010.

[8] David Challener, Kent Yoder, Ryan Catherman, David Safford, and Leendert Van Doorn. *A practical guide to trusted computing.* IBM press, 2007.

[9] Benjie Chen and Robert Morris. Certifying program execution with secure processors. In *HotOS*, pages 133–138, 2003.

[10] Cheun Ngen Chong, Zhonghong Peng, and Pieter H Hartel. Secure audit logging with tamper-resistant hardware. *SEC*, 250:73–84, 2003.

[11] Whitfield Diffie, Paul C Van Oorschot, and Michael J Wiener. Authentication and authenticated key exchanges. *Designs, Codes and cryptography*, 2(2):107–125, 1992.

[12] NIST FIPS. 198-1: The keyed-hash message authentication code (hmac). *National Institute of Standards and Technology, Federal Information Processing Standards*, 2008.

[13] Eiichiro Fujisaki and Tatsuaki Okamoto. Statistical zero knowledge protocols to prove modular polynomial relations. In *Advances in Cryptology—CRYPTO'97*, pages 16–30. Springer, 1997.

[14] Rainer Gerhards et al. Rfc 5424: The syslog protocol. *Request for Comments, IETF*, 2009.

[15] TRUSTED COMPUTING GROUP et al. Trusted computing group home page, 2009.

[16] Louis C Guillou and Jean-Jacques Quisquater. A "paradoxical" identity-based signature scheme resulting from zero-knowledge. In *Proceedings on Advances in cryptology*, pages 216–231. Springer-Verlag New York, Inc., 1990.

[17] 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*, pages 203–211. Australian Computer Society, Inc., 2006.

[18] Gene Itkis and Leonid Reyzin. Forward-secure signatures with optimal signing and verifying. In *Advances in Cryptology—Crypto 2001*, pages 332–354. Springer, 2001.

[19] J Jonsson and B Kaliski. Public-key cryptography standards (pkcs)# 1: Rsa cryptography specifications version 2.1 (rfc 3447). *Internet Engineering Task Force (IETF)*, 2003.

[20] Karen Kent and Murugiah Souppaya. Guide to computer security log management. *NIST special publication*, pages 800–92, 2006.

[21] Anton Kozlov and Leonid Reyzin. Forward-secure signatures with fast key update. In *Security in communication Networks*, pages 241–256. Springer, 2003.

[22] Chris Lonvick. Rfc 3164: The bsd syslog protocol. 2001.

[23] Di Ma. Practical forward secure sequential aggregate signatures. In *Proceedings of the 2008 ACM symposium on Information, computer and communications security*, pages 341–352. ACM, 2008.

[24] Di Ma and Gene Tsudik. Forward-secure sequential aggregate authentication. In *IEEE Symposium on Security and Privacy*, pages 86–91, 2007.

[25] TPM Main. Part 2 tpm structures. *Specification version 1.2*, 1, 2007.

[26] 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*, pages 400–417. Springer, 2002.

[27] Jonathan M McCune, Bryan J Parno, and Adrian Perrig. Flicker: Minimal tcb code execution. `https://sparrow.ece.cmu.edu/group/flicker.html`, 2011. [Online; accessed 7-March-2014].

[28] Jonathan M McCune, Bryan J Parno, Adrian Perrig, Michael K Reiter, and Hiroshi Isozaki. Flicker: An execution infrastructure for tcb minimization. In *ACM SIGOPS Operating Systems Review*, volume 42, pages 315–328. ACM, 2008.

[29] Fuyou Miao, Yuzhi Ma, and Joseph Salowey. Transport layer security (tls) transport mapping for syslog. In *RFC*, 2009.

[30] Darren New and Marshall T Rose. Rfc 3195: Reliable delivery for syslog. 2001.

[31] Magnus Nyström, Martin Nicholes, and Vincent J Zimmer. Uefi networking and pre-os security. *Intel Technology Journal*, 15(1), 2011.

[32] T Polk, L Chen, S Turner, and P Hoffman. Rfc 6194: Security considerations for the sha-0 and sha-1 message-digest algorithms. *Internet Engineering Task Force (IETF)*, 2011.

[33] Jerome H Saltzer, David P Reed, and David D Clark. End-to-end arguments in system design. *ACM Transactions on Computer Systems (TOCS)*, 2(4):277–288, 1984.

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

[35] Arvind Seshadri, Mark Luk, Elaine Shi, Adrian Perrig, Leendert van Doorn, and Pradeep Khosla. Pioneer: verifying code integrity and enforcing untampered code execution on legacy systems. *ACM SIGOPS Operating Systems Review*, 39(5):1–16, 2005.

[36] Elaine Shi, Adrian Perrig, and Leendert Van Doorn. Bind: A fine-grained attestation service for secure distributed systems. In *Security and Privacy, 2005 IEEE Symposium on*, pages 154–168. IEEE, 2005.

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

[38] Xiaoyun Wang, Yiqun Lisa Yin, and Hongbo Yu. Finding collisions in the full sha-1. In *Advances in Cryptology–CRYPTO 2005*, pages 17–36. Springer, 2005.