Preserving Privacy in Transparency Logging

Tobias Pulls

Faculty of Health, Science and Technology

Computer Science

DISSERTATION | Karlstad University Studies | 2015:28

Preserving Privacy in Transparency Logging

Tobias Pulls

DISSERTATION | Karlstad University Studies | 2015:28

Preserving Privacy in Transparency Logging

Tobias Pulls

DISSERTATION

Karlstad University Studies | 2015:28

urn:nbn:se:kau:diva-35918

ISSN 1403-8099

ISBN 978-91-7063-644-8

Distribution:
Karlstad University
Faculty of Health, Science and Technology
Department of Mathematics and Computer Science
SE-651 88 Karlstad, Sweden
+46 54 700 10 00

Print: Universitetstryckeriet, Karlstad 2015

WWW.KAU.SE

Abstract

The subject of this dissertation is the construction of privacy-enhancing technologies (PETs) for transparency logging, a technology at the intersection of privacy, transparency, and accountability. Transparency logging facilitates the transportation of data from service providers to users of services and is therefore a key enabler for ex-post transparency-enhancing tools (TETs). Ex-post transparency provides information to users about how their personal data have been processed by service providers, and is a prerequisite for accountability: you cannot hold a service provider accountable for what is unknown.

We present three generations of PETs for transparency logging to which we contributed. We start with early work that defined the setting as a foundation and build upon it to increase both the privacy protections and the utility of the data sent through transparency logging. Our contributions include the first provably secure privacy-preserving transparency logging scheme and a forward-secure append-only persistent authenticated data structure tailored to the transparency logging setting. Applications of our work range from notifications and deriving data disclosures for the Data Track (an ex-post TET) to secure evidence storage.

In memory of Tommy Andersson

Acknowledgements

Early during my time as a PhD student Hans Hedbom played a pivotal role in starting our work on privacy-preserving transparency logging. Hans also introduced me to Karel Wouters who had done related work with complementary properties. Karel took over where Hans left off, and together we laid the foundation for the work presented in this dissertation. Karel brought Roel Peeters into the mix, and with Roel's help we matured privacy-preserving transparency logging. Roel and I continued collaborating, in the end producing the majority of the work presented here. Thank you Roel, Karel, and Hans for all the fruitful collaboration over the years. Without you, this dissertation would not have been possible.

Throughout my studies my supervisor Simone Fischer-Hübner and my co-supervisor Stefan Lindskog have always offered support and guidance. Thank you for all that you have done and for giving me the opportunity to pursue a PhD.

My colleagues at the Department of Mathematics and Computer Science have provided a wonderful working environment over the years. I would especially like to thank the members of the privacy and security research group, including honorary member Julio Angulo.

To my family and friends, outside and at work, thank you for all of your support. I am in your debt. Susanne, thank you for all the "Långa Nätter".

Finally, I would also like to thank the EU FP7 projects A4Cloud and PrimeLife, Google, and the Swedish Knowledge Foundation for funding my research. Collaborating with both academia and industry have provided valuable experiences during my studies.

Karlstad, May 2015 Tobias Pulls

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

> We kill people based on metadata.
>
> General *Michael Hayden* – former director of the NSA and the CIA

Metadata is data about data. Personal data is data related to a person. Metadata about personal data is also personal data, i.e., data about data that potentially reveals information about individuals may reveal more information about individuals [47]. Take, for instance, statistics about the average socioeconomic status of inhabitants in different regions of a country. Given an individual's address, which is clearly personal data, the statistics (metadata about an address) enable the inference of socioeconomic status of the individual with some probability. Knowledge of the world enables the most seemingly trivial piece of data to be the signal that links an individual to a context outside of the individual's control. It is like the butterfly effect, but for data; e.g., sending a message to the wrong number can cause seemingly disproportionate events that results in one or more individuals' deaths. Privacy matters, because metadata might get you killed tomorrow by largely unaccountable organisations, as the introductory quote makes clear.

Transparency, like privacy, plays an important role in society. Transparency assists in holding organisations and governments accountable, and it is also a key privacy principle to inform people when they attempt to exercise control over their personal spheres. The goal of this dissertation is the construction of a privacy-preserving technological tool for enhancing transparency. The tool can be used as a building

block for making entities accountable for their actions. We strive to prevent the creation of new metadata as a consequence of using our tool. We do this to protect the privacy of the users of the tool and any other persons whose actions are being made transparent, and because Transparency-Enhancing Tools (TETs) that create new metadata are less efficient at their core task.

1.1 The Ethos of PETs

Privacy-Enhancing Technologies (PETs) aim to assist persons in protecting their privacy through technological means. Prime examples of PETs are the deployed and widely used low-latency anonymity network Tor [45], and the anonymous credentials system idemix [33]. Van Blarkom *et al.* [115] define PETs broadly as:

> *"PET stands for a coherent system of ICT measures that protects privacy by eliminating or reducing personal data or by preventing unnecessary and/or undesired processing of personal data, all without losing the functionality of the information system."*

The definition is broad, covering "unnecessary" and "undesired" processing of personal data. What is strictly necessary from a technical perspective is often surprising, as powerful technologies like idemix make apparent. Take, e.g., a service that is required by law to verify that its visitors are at least 18 years old. Intuitively, such a service by necessity must learn the age of its visitors. However, this is not the case. PETs like idemix can produce a zero-knowledge proof that cryptographically proves that a visitor is at least 18 years old, as certified by some trusted party on an idemix credential, without revealing the actual age. Preventing "undesired" processing is closely related to the wishes of the person whose personal data is subject to processing, also covering technologies that informs persons of the consequences of data processing or data disclosure. Therefore, technology that informs persons that revealing ones age to a service is not technically necessary may also be considered a PET, since it may prevent the person from disclosing the data or sway his or her desires.

Diaz *et al.* [44] highlights the stark difference in the view on privacy between a stricter definition of PETs as technology to protect persons from *surveillance* and legal privacy frameworks like the EU Data Protection Directive 95/46/EC. Legal privacy frameworks promote the model of a *trustworthy controller*, i.e., the entity in control (in the legal and potentially technical sense) of personal data is assumed trustworthy in the sense that it intends to comply with legal obligations. The controller is required

to act as an accountable steward of personal data, aiming to fulfill its legal obligations and taking the subject's best interests into account with regard to privacy as defined in the legal privacy frameworks (which may or may not coincide with a person's expectations of privacy). Tools that assist the controller in performing this role may be seen as *accountability tools*. A strict definition of PETs against surveillance is fundamentally different, in that PETs view the controller as a potential *adversary* with regard to privacy. The controller may be trusted to provide a service, but it is not trusted or assumed to act with the privacy of its users in mind. The controller is just as likely, if not more likely due to its privileged position, to attempt to *surveil* persons, just like other organisations or governments. PETs should protect people from such surveillance, regardless of adversary. Goldberg *et al.* [55] succinctly summarises the ethos of PETs by paraphrasing the cypherpunks credo as:

> *"Privacy through technology, not through legislation."*

1.2 TETs and Transparency Logging

Transparency-Enhancing Tools (TETs) are at their core tools that provide persons with data that concerns their privacy [59]. Unlike strict definitions of PETs in general, TETs focus on providing data, while strict PETs hide data. TETs take the form of both *technological* and *legal* tools. Legal tools, like the EU Data Protection Directive 95/46/EC, give persons access to their personal data at controllers and specifies data to be shared, e.g., when further personal data is obtained from third-parties (Articles 10–12). Technological tools may assist persons with *counter profiling*, providing insights to persons about how their personal data is processed, and what the consequences of the processing may be [49]. In a sense, TETs can make profiling transparent, such that persons become aware of actions that they may otherwise be unaware of. A prime example of such a technological TET is Mozilla Lightbeam[1]. Lightbeam is a Firefox add-on that visualises relationships between sites and their embedded links to third-party sites as you browse the web. The tool in a sense counter-profiles sites that profile their visitors. The resulting visualisation makes otherwise opaque practices transparent to previously potentially unaware persons.

TETs can be classified into ex-ante and ex-post TETs. Ex-ante TETs provide transparency prior to a user disclosing data to a controller, while ex-post TETs provide transparency after data disclosure. This makes Mozilla Lightbeam an ex-post

[1] https://www.mozilla.org/en-US/lightbeam/, accessed 2015-03-12.

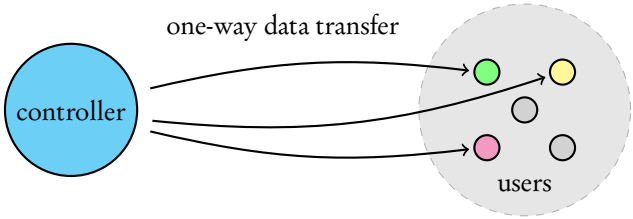


Figure 1.1 The conceptual setting for transparency logging, where a controller transfers data to one or more users concerning the controller's use of their personal data.

TET. A good example of ex-ante TETs are privacy policy languages and tools for their visualisation prior to the user disclosing data to a controller, with the goal of informing users of the consequences of data disclosure [10]. In general, privacy issues are more prominent in ex-post TETs, since the user has disclosed data to a controller and any subsequent actions of the user (of a TET acting on his or her behalf) may be linked to the data disclosure and reveal additional information about the user.

This dissertation is focused on a technological ex-post TET: *transparency logging*. The need for transparency logging was clearly identified in the Future of Identity in the Information Society (FIDIS) network funded by the EU's 6th Framework Programme. To perform counter-profiling and successfully anticipate consequences of personal data processing, TETs need insight into the actual processing being done by controllers [49]. Transparency logging is a TET for transporting this processing data. Transparency logging therefore dictates at least two players in its setting: a *controller*, e.g., of a service, that generates data intended for *users*, where the data describes processing on the user's personal data, so the users are data subjects. Fundamentally, the transfer is *one-way*, from the controller to the users. Figure 1.1 illustrates the conceptual transparency logging setting.

The transfer of data in transparency logging facilitates a knowledge transfer from the controller to the users. This presumably reduces the *information asymmetry* between the controller and users, enabling another TET to counter profile the controller. Our work has focused on how to enable this knowledge transfer in a privacy-preserving way: in other words, this dissertation focuses on the construction of *PETs for transparency logging*. We note that if the data transfer leaks information to the controller about the users, then transparency logging can actually *increase* the information asymmetry between the parties. Protecting users' privacy in transparency

logging is therefore a key activity for the tool to have its desired effect in conjunction with a counter-profiling TET.

In a sense, in this dissertation we develop strict PETs for an accountability tool in the form of transparency logging, which is a key mechanism for efficient TETs. In another sense, embracing a more broad definition of PETs that covers both accountability tools and TETs, we simply develop efficient PETs for making data processing transparent. Regardless of the viewpoint, we stress that we focus on the technical design of privacy-preserving transparency logging schemes, and not on what should be logged, or how this data should be used to, e.g., inform persons or hold services accountable.

1.3 Research Questions

This dissertation explores the following two research questions in relation to privacy-preserving transparency logging:

RQ1 *What are relevant security and privacy requirements for transparency logging?*
That TETs in general pose a threat to privacy may appear counter intuitive, since an explicit goal of TETs is to provide transparency, which is a key privacy principle. However, the means to provide transparency is the transfer of data that are, or are related to, personal data. The mechanism for data transfer, just like any other means of communication, poses a potential threat to privacy. Chapter 2 of this dissertation is focused on defining requirements for transparency logging. The requirements are based upon a number of identified security and privacy considerations derived from early work on transparency logging. Chapter 3 provides the first formal definitions of several privacy properties in the setting and identifies key security properties from related work. Chapter 4 further elaborates on requirements, and Chapters 5 and 6 provide new derived formal definitions.

RQ2 *How can transparency logging be designed to preserve privacy?*
Given the identified requirements from RQ1, this research question explores how TETs can be designed to preserve privacy. As noted in Section 1.2, preserving privacy in TETs is a fundamental part of ensuring that the TETs function optimally. This is the main focus of this dissertation for transparency logging. Chapter 2 starts by summarising early work on privacy-preserving transparency logging. Chapter 3 presents a provably secure privacy-preserving transparency

logging scheme. Chapter 4 identifies the need for a key building-block for improving privacy, and Chapter 5 presents such a building block. Chapter 6 presents the culmination of the work in this dissertation, which is an updated design of a privacy-preserving transparency logging scheme. Finally, Chapter 7 presents two applications of the scheme in Chapter 6, one of which is for the Data Track, a TET.

1.4 Research Methods

We use the *scientific* and *mathematical* research methods [54, 90]. Both methods have in common that they iteratively identify a question, analyse the question and propose an answer, gather and analyse evidence to evaluate the validity of the proposed answer, and finally review the outcome of the previous steps before starting over again. The mathematical method is in the setting of formal mathematical models, which are (at times) abstractions of the real natural world. The real natural world is the exclusive setting of the scientific method, which studies the natural world, at times with the help of mathematics [54]. Computer science is inherently mathematical in its nature [113], but deals also with the application of this theory in the real world, which makes it a science [46]. Like computer science, *cryptography* is a field that shares connections to both the scientific and mathematical research methods.

The field of cryptography can conceptually be split into two: theoretical and applied cryptography. Theoretical cryptography is mathematical in its nature, putting emphasis on formal definitions and proofs within mathematical models, perhaps most notably using the method of *provable security* [57]. Succinctly, the goal of provable security is to prove a *reduction* of breaking a cryptographic scheme (like a protocol) to solving a computational problem (assumed to be hard) or breaking a primitive (like a cryptographic hash function). In other words, you prove that if you can break the cryptographic scheme, you can solve the computational problem or break the cryptographic primitive (with some probability within a period of time). When making this proof, it is essential to define (i) what "breaking" means, i.e., what is the *goal of the adversary*, and (ii) what are the *capabilities* of the adversary, i.e., what is the *adversary model*.

Applied cryptography bridges the mathematical model of theoretical cryptography and the real natural world by applying cryptography in real-world computer systems and networks, often with the goal of *securing* systems and communication. While few question the value of provable security (and fundamental research in gen-

eral) in and of itself in theoretical cryptography, the value of the method for applied cryptography is a hot topic of discussion [41, 56, 68, 69, 99]. Computer systems are unfathomably complex, and for securing systems with cryptography both cryptanalysis and sound engineering practices also play an important role. This is especially telling when comparing work done in the cryptography community with that done in, .e.g, the PETs community (see, .e.g., the work underlying idemix [33] as compared to Tor [45]). Baring a systematic (some even go as far as saying scientific approach [56]) to securing systems, we end up in a never-ending "propose and break" cycle.

The work in this dissertation sought to balance the wish for practical research prototypes (as a consequence of the work taking place in research projects dictating workable prototypes and their integration) with the rigour associated with provable security where appropriate. This dissertation starts as largely *exploratory*, involving *literature review* in areas closely related to privacy-preserving transparency logging, with the goal of defining the setting and relevant functional, security, and privacy requirements. In other words, we start by *defining the problems* to tackle. With a better idea of the setting and requirements (the problems to tackle), we produce privacy and security definitions for the setting (what the adversary wants to break), proposed a cryptographic scheme based on prior joint work (our solution), and analyse how well our proposed scheme fulfills the definitions. Our analysis includes proofs for different properties based on our definitions that have *varying degrees of rigour*, depending on how well-defined and closely related the properties are to prior work in related settings. Next, we review the current state of the first version of our privacy-preserving transparency logging scheme, identifying limitations and potential paths forward. This informs us to ask better *questions*, enabling us to create a cryptographic scheme with new and modified privacy and security definitions, following largely the same method as for the first proposed solution. For both versions of the privacy-preserving transparency logging schemes, we have *experimentally* evaluated the performance, with a focus on entry generation, by *proof-of-concept* implementations. For the authenticated data structure Balloon, in Chapter 5, we analytically compared the expected complexity to ease comparison with a related data structure in addition to experimental evaluation of performance.

1.5 Summary of Related Work

The earliest work, to our knowledge, on using cryptographic schemes from the secure logging area to protect systems that attempt to make data processing transparent

is by Sackmann *et al.* [103]. Trusted auditors use secure logs of data processing as so called "privacy evidence" to compare the actual processing with the processing stated in a privacy policy that users have consented to. Wouters *et al.* [122] and Hedbom *et al.* [62][2] look at transparency logging schemes from another angle than the work of Sackmann et al.: users take on the primary role of auditors of the logged data that relates to them, removing the need for trusted auditors. Since log entries now relate to different users who are actively participating in the scheme, the setting becomes user-centric. This new setting leads to new privacy threats, in addition to the potential threats to privacy posed by the actual contents of log entries. Wouters *et al.* address the linkability issue between logs at different log servers in a distributed setting, while Hedbom *et al.* address the linkability between log entries in one log at one log server. These schemes, including the schemes in this dissertation, build upon the secure logging system by Schneier and Kelsey [104]. Table 1.1 compares the different schemes with regard to their security and privacy properties. The properties are deletion-detection forward integrity (DDFI), secrecy (S), forward unlinkability of users and log entries (FUE), forward unlinkability of user identifiers (FUI), and support for distributed settings. Chapters 2 and 3 explain these properties and schemes more thoroughly.

Table 1.1 Comparison of privacy and security properties with related work.

Properties	DDFI	S	FUE	FUI	Distributed
Hedbom *et al.* [62]	✓	✓	≈		
Sackmann *et al.* [103]	≈	✓			
Schneier and Kelsey [104]	≈	✓			
Wouters *et al.* [122]		✓		✓	✓
This Dissertation	✓	✓	✓	✓	✓

In Chapter 5 we present a customised authenticated data structure for privacy-preserving transparency logging named Balloon. Balloon is closely related to persistent authenticated dictionaries (PADs) [8]. Thanks to being append-only, instead of also supporting deletion like PADs, Balloon is more efficient. Table 1.2 highlights the difference in expected complexity between Balloon and the most efficient tree-based PAD construction in practice according to Crosby & Wallach [40]. Chapter 5 provides further details.

[2]Note that I contributed to the work of Hedbom *et al.* [62] before starting my PhD studies.

Table 1.2 The differences in expected complexity between Balloon and an efficient PAD construction [40], where n is the number of entries and v the size of the version cache.

Expected Complexity	Total Storage Size (controller)	Query Time (past)	Insert Storage Size (server)	Insert Proof Size
Balloon	$\mathcal{O}(1)$	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$	$\mathcal{O}(\log n)$
Tree-based PAD [40]	$\mathcal{O}(n)$	$\mathcal{O}(\log v \cdot \log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$

By taking privacy-preserving transparency logging in the direction of using authenticated data structures, the Insynd scheme presented in Chapter 6 provides (in addition to the properties in Table 1.1) stronger privacy protections and several types of *public verifiability*. Publicly verifiable deletion-detection forward integrity are provided by secure logging schemes like Logcrypt by Holt [64] and one of the schemes of Ma and Tsudik [75]. In addition to this property, Insynd also enables the creation of publicly verifiable proofs of controller ("who is the controller?"), user ("who is the user?"), message ("what was the message?"), and time ("when was this sent?") with limited negative impact on privacy. This greatly increases the utility of all data sent through our scheme. Earlier work either lacked this ability completely, like Wouters *et al.* [122], or only provided it by revealing private keys with potentially huge negative privacy implications, like Hedbom *et al.* [62].

1.6 Contributions

This dissertation makes the following six contributions:

C1 *A thorough definition of the privacy-preserving transparency logging setting.* While the FIDIS network clearly identified the need for transparency logging by controllers for TETs, prior to our work the setting was largely undefined. In Chapter 2, we identify all the relevant players in the setting and move towards proper definitions of privacy and security with regard to the players, extending the work of Wouters *et al.* [122] and Hedbom *et al.* [62]. Chapter 4 further redefines the setting, restricting the role of a key player. Finally, Chapter 6 defines the strongest possible practical adversarial setting for transparency logging (in terms of the controller and server) and links the problems in the setting with those in the secure messaging area.

C2 *Definitions of privacy and security properties for transparency logging.*
As a more concrete setting emerged during our work, we could define what it means for a transparency logging scheme to be privacy preserving and secure. We identified relevant definitions from related areas, like secure logging, and made our own definitions where we found gaps. Chapter 2 identifies a large number of privacy and security requirements in the setting, which are turned into a few formal definitions in Chapter 3. Chapters 4 and 6 further refine the definitions, and introduce some new, as we evolve the setting. We define secrecy, deletion-detection forward integrity, forward unlinkability of entries, publicly verifiable consistency, publicly verifiable proofs (of controller, time, user, and message), and forward unlinkability of user identifiers in the transparency logging setting.

C3 *A cryptographic scheme for distributed privacy-preserving transparency logging.*
Chapter 3 presents the first provably secure privacy-preserving transparency logging scheme. The scheme supports distributed settings, which is an important aspect of the transparency logging setting. We introduce formal definitions of security and privacy, and prove that our proposed scheme provides the defined properties in the random oracle model.

C4 *A forward-secure append-only persistent authenticated data structure.*
In Chapter 4, we identify the need for an authenticated data structure to relax the trust assumptions for transparency logging, and thereby support a stronger adversary model. We present Balloon, a forward-secure append-only persistent authenticated data structure in Chapter 5. Balloon is tailored to the transparency logging setting, and therefore provides significantly better performance than related work from other settings. We evaluate Balloon in the standard model.

C5 *A privacy-preserving one-way messaging scheme.*
Chapter 6 presents a privacy-preserving one-way messaging scheme named Insynd that uses Balloon as a building block. Insynd has all privacy and security properties of the transparency logging scheme presented in Chapter 3 in a stronger adversary model. The majority of properties are sketched in the random oracle model. The scheme also provides several *publicly verifiable* properties, perhaps most notably the ability to create proofs of user, controller, message, and time of existence. These proofs greatly increase the utility of all data sent through the scheme.

C6 *Applications for transparency logging schemes.*
Chapter 7 presents two applications of transparency logging schemes. One application is a textbook example of using the transparency logging scheme to move data from controllers to users using a TET for counter-profiling of data disclosures and processing. The other application is the use of the Insynd scheme as an *evidence store* for an audit agent system. Chapter 3 also describes how a transparency logging scheme can be used to make access to electronic health records transparent to patients. Last, but not least, Chapter 6 shows that Insynd can be used as a traditional secure logging system with comparable performance to the state-of-the-art secure logging scheme PillarBox [31], while still providing additional properties like publicly verifiable consistency and publicly verifiable proofs.

1.7 Outline

This dissertation is structured as follows. Chapter 2 describes the start of others and our work on defining the setting of transparency logging, and important related security and privacy requirements and properties. Chapter 3 presents the cryptographic scheme that resulted from the early work described in Chapter 2. Chapter 4 presents a number of limitations in the scheme proposed in Chapter 3, an approach to solving some of them, and a motivation for the direction moving forward. Chapter 5 presents Balloon: a forward-secure append-only persistent authenticated data structure. This is a key building-block in addressing several of the short-comings identified in Chapter 4. Chapter 6 presents Insynd: a privacy-preserving secure one-way messaging scheme using balloons. This scheme uses Balloon, as presented in Chapter 5, to construct an improved version of the scheme from Chapter 3. Chapter 7 provides a high-level description of two applications of Insynd from the EU FP7 project A4Cloud. Finally, Chapter 8 concludes this dissertation with some final remarks. Appendix A contains a list of all publications I have contributed to over the course of my PhD studies.

1.8 Sources

Before the next chapter we present the sources of this dissertation roughly in the order of their use. For each source, I state my contribution and that of my co-authors.

[88] R. Peeters, T. Pulls, and K. Wouters. Enhancing transparency with distributed privacy-preserving logging. In *ISSE 2013 Securing Electronic Business Processes*, pages 61–71. Springer, 2013

For [88], Peeters was the driving force while Wouters and I provided feedback. This publication makes up part of Chapter 1.

[60] H. Hedbom and T. Pulls. Unlinking database entries: Implementation issues in privacy preserving secure logging. In *2010 2nd International Workshop on Security and Communication Networks (IWSCN)*, pages 1–7. IEEE, 2010

[61] H. Hedbom, T. Pulls, and M. Hansen. Transparency tools. In J. Camenisch, S. Fischer-Hübner, and K. Rannenberg, editors, *Privacy and Identity Management for Life*, pages 135–143. Springer, 2011

[62] H. Hedbom, T. Pulls, P. Hjärtquist, and A. Lavén. Adding secure transparency logging to the PRIME Core. In M. Bezzi, P. Duquenoy, S. Fischer-Hübner, M. Hansen, and G. Zhang, editors, *Privacy and Identity Management for Life*, volume 320 of *IFIP Advances in Information and Communication Technology*, pages 299–314. Springer Boston, 2010

For [60–62], Hedbom was the driving force behind writing the papers and book chapter. We jointly worked on defining the problems and the proposed solutions. Hansen, in [61], contributed to a legal analysis not used in this dissertation. Hjärtquist and Lavén, in [62], provided thorough feedback. Chapter 2 uses material from these publications.

[97] T. Pulls, K. Wouters, J. Vliegen, and C. Grahn. Distributed Privacy-Preserving Log Trails. Technical Report 2012:24, Karlstad University, Department of Computer Science, 2012

For [97], I collaborated with Wouters on the bulk of the work. I came up with the idea of forking and wrote all algorithms for defining the non-auditable system. Wouters made the system auditable and contributed with a thorough related work section on secure logging together with a description of the early work in Wouters *et al.* [122]. Vliegen and Grahn contributed with a description of their respective proof of concept hardware and software implementations. Chapters 2–4 use material from this publication.

[95] T. Pulls, R. Peeters, and K. Wouters. Distributed privacy-preserving transparency logging. In *Proceedings of the 12th annual ACM Workshop on Privacy in the Electronic Society, WPES 2013, Berlin, Germany, November 4, 2013*, pages 83–94. ACM, 2013

For [95], I and Peeters were the driving force behind the publication. We worked on the definitions and scheme together. Peeters focused on the evaluation and Wouters provided detailed feedback. This publication makes up the majority of Chapter 3.

[118] J. Vliegen, K. Wouters, C. Grahn, and T. Pulls. Hardware strengthening a distributed logging scheme. In *15th Euromicro Conference on Digital System Design, DSD 2012, Cesme, Izmir, Turkey, September 5-8, 2012*, pages 171–176. IEEE Computer Society, 2012

Vliegen did the vast majority of the work on the hardware for [118]. Grahn implemented the software connector to the hardware. Wouters and I contributed on the sketched changes to the cryptographic scheme. Chapter 4 uses material from this publication.

[93] T. Pulls and R. Peeters. Balloon: A forward-secure append-only persistent authenticated data structure. Cryptology ePrint Archive, Report 2015/007, 2015

For [93], it was my idea to look at authenticated data structures in general and I came up with the initial construction. I closely collaborated with Peeters on all parts of the paper. This publication makes up the majority of Chapter 5.

[94] T. Pulls and R. Peeters. Insynd: Secure one-way messaging through Balloons. Cryptology ePrint Archive, Report 2015/150, 2015

For [94], I closely collaborated with Peeters on all parts of the paper. This publication makes up the majority of Chapter 6.

[11] J. Angulo, S. Fischer-Hübner, T. Pulls, and E. Wästlund. Usable transparency with the data track: A tool for visualizing data disclosures. In *Proceedings of the 33rd Annual ACM Conference Extended Abstracts on Human Factors in Computing Systems*, CHI '15, pages 1803–1808, Seoul, Republic of Korea, 2015. ACM

For [11], Angulo was the driving force. Fischer-Hübner contributed with the early concepts and design, and Wästlund participated in the evaluation of the usability tests of the design. I contributed with some of the earlier designs on the trace view as well as the backend of the Data Track. Chapter 7 uses material from this publication.

[100] T. Ruebsamen, T. Pulls, and C. Reich. Secure evidence collection and storage for cloud accountability audits. CLOSER 2015 - Proceedings of the 5th International Conference on Cloud Computing and Services Science, Lisbon, Portugal, May 20-22, 2015

For [100], Ruebsamen was the driving force together with Reich on their Audit Agent System. I contributed with the details on Insynd and on how to accomplish the integration. Chapter 7 uses material from this publication.

Chapter 2

Towards Defining Privacy for Transparency Logging

> Technology is neither good nor bad; nor is it neutral.
>
> *Melvin Kranzberg* – The first law of technology

We start by summarising early work related to privacy-preserving transparency logging in Sections 2.1–2.3, identifying key contributions and limitations. I was part of the work summarised in Sections 2.1 and 2.2, while Section 2.3 summarises the work of Wouters *et al.* [122]. Next, in Section 2.4, we take the early work and derive from it the setting for transparency logging. Given the setting, Section 2.5 derives a number of requirements, identified as part of earlier work. We conclude the chapter in Section 2.6.

2.1 Privacy-Preserving Secure Logging

In general, a log can be viewed as a record of sequential data. A secure log, when compared to a standard log, protects the confidentiality and integrity of the entries in the log [104]. A privacy-preserving secure log, in addition to being a secure log, tries to address the privacy problems related to the fact that it stores a log of how

personal data is processed. Each entry in the privacy-preserving secure log concerns one *data subject*; the entity on whose behalf the entry is made. We refer to the entity performing the data processing and generating the log as the *data processor*.

The log is secure in the sense that confidentiality is provided by encrypting the data stored in entries and integrity is provided, by using, e.g., hash functions and message authentication codes. The log is privacy preserving by providing the following properties:

1. The data in a log entry is encrypted under the public key of the data subject the data relates to. This ensures that no other entity can read the data stored in the log, which could violate the privacy of the data subject.

2. Data subjects and the data processor can independently validate the integrity of the entries that concern them. If multiple entities are needed to validate the integrity of parts of the log, no single entity will fully trust the integrity of, and hence the data stored in, entries of the log.

3. Unlinkability of data subjects and their entries; that is, to which data subject an entry relates is hidden. Without this property, it would be possible to tell how many entries there are in the log for each data subject, which might reveal, for example, how frequent a data subject is using a service (this is of course highly dependent on what is stored in the log).

4. The data processor safely provides anonymous access to the log entries for data subjects. Requiring authenticated access to entries could allow an attacker to link entries to data subjects, depending on the authentication mechanism.

2.1.1 Adversary Model

Privacy-preserving secure logging has its roots in the secure logging system by Schneier and Kelsey [104], and inherits its adversary model. The privacy-preserving secure log provides the properties described previously for all entries committed to the log prior to an adversary compromising the data processor's system. Once compromised, little can be done to secure or provide any privacy to future entries committed to the log. The adversary model is thus that the data processor is initially trusted and then at some point in time becomes compromised (forward security [19]). We consider the network benign due to scope, but note that data subjects will break the unlinkability property if they do not use an anonymity network, such as Tor [45], when retrieving their log entries.

2.1.2 Overview of Our Idea

Figure 2.1 gives an overview of the steps involved by the data processor from performing data processing to creating a new log entry for the log. The transformation of the description into a log entry is done by a series of cryptographic operations involving a hash function, an asymmetric encryption algorithm, a signature algorithm, and a message-authentication code (MAC). The transformation operates on the current *state* of the data processor. The state contains several *authentication keys* that are continuously evolved as entries are generated. There is one key per data subject and one for the data processor. Figure 2.2 shows how state is updated for the subject as part of the transformation for one entry. The state transformation is just like a *ratchet* in the sense that it can only be turned one-way. Each update to the state overwrites the prior state.

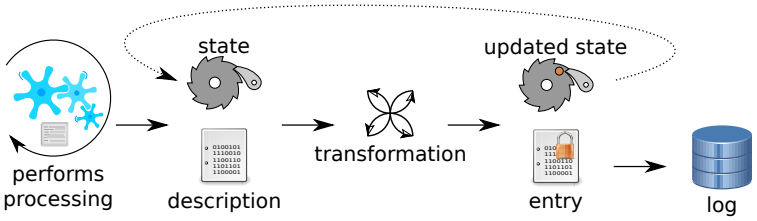


Figure 2.1 Overview of the steps from data processing to log entry by the data processor.

The state transformation generates two fields as part of the log entry: an identifier (ID) and a data chain (DC). The identifier identifies the entry for the subject. The subject can generate all identifiers deterministically from the initial authentication key (AK_0), agreed upon as part of setting up the privacy-preserving secure log. The data chain is a MAC that covers all entries generated for the subject. Since the MAC is keyed with the current authentication key, which is overwritten by applying a hash function as part of the state transformation, any data chains already generated cannot be altered by an adversary who compromises the processor. As for the subject, there is state for the processor that is used to generate an identifier and data chain for the processor for each entry. This enables the processor to order entries (based on identifier generation) and to verify the integrity of all entries.

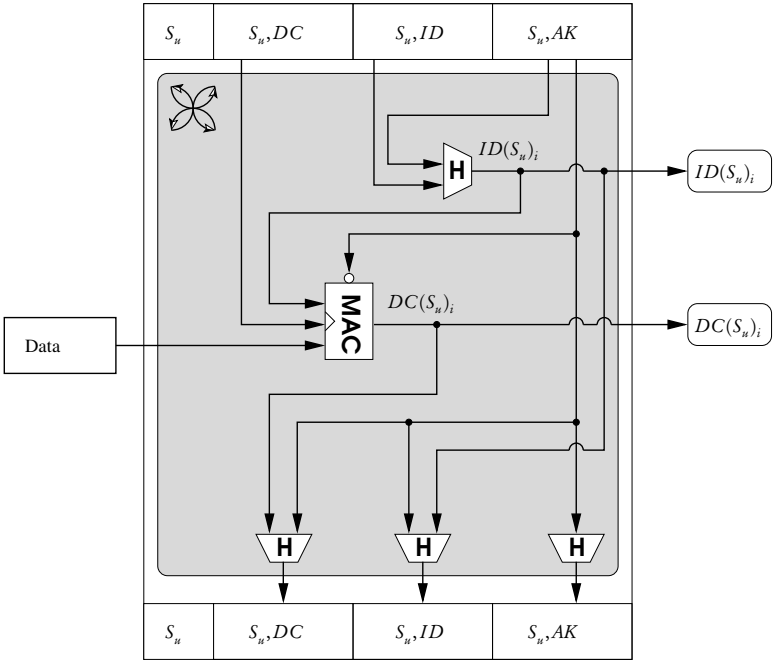


Figure 2.2 State overview and update for one entity.

2.1.3 Key Contributions and Limitations

The key contributions, as far as PETs for transparency logging are concerned, from privacy-preserving secure logging are:

State for data subjects Keeping and evolving state for each data subject. Inspired by constructions from the secure logging area.

Unlinkability of entries and data subjects A key privacy property in the setting.

We note several limitations with privacy-preserving secure logging for the purpose of transparency logging:

Only one data processor We are limited to one data processor only. If the data processor shares the data subject's data with another data processor there is currently no way for the logging to continue without having an online data subject repeating the setup protocol (not presented in this overview). This

prohibits the system from being used for logging processes across multiple data processors.

No external auditing The auditing capabilities in the scheme are limited to the data subjects and data processor being able to verify the integrity of the log entries for themselves, with no support of external auditing.

The data processor stores the log The log functionality is hosted by the data processor itself, which requires it to store the log entries and to open up an interface towards the data subjects, to inspect their log entries. For the data processor, it might be interesting to be able to outsource these services.

Next, we look at implementation issue related to storing entries for privacy-preserving secure logging, before moving on to another related logging scheme.

2.2 Unlinking Database Entries

One distinguishing property of the privacy-preserving secure logging scheme is that data subjects and their log entries are *unlinkable* as a consequence of how log entries are generated. In the model used for privacy-preserving secure logging, the storage of log entries is seen as a *set*. This is hard to achieve in practice. When an adversary compromises the data processor, if it wishes to link previously generated log entries to subjects, it will use all information at its disposal in practice. If log entries can be ordered based on when they were generated (i.e., the log can be seen as a chronologically sorted list) due to implementation flaws, then an adversary may use this information and correlate it with other sources, such as access logs that record events that trigger data processing for a particular data subject, and therefore results in log entries.

External logs, like access logs, will most likely be present in the majority of modern computer systems created by applications or by the operating system itself. An example to highlight the problem would be a log keeping a detailed record of how personal data given to a social networking site is processed and handed out to a web application. The web application is running on the Apache web server, which keeps a detailed access log of all requests. Imagine the following entry in the access log: "127.0.0.1 – bob [23/Feb/2015:13:55:36 -0700] "*GET /profile.php?id=5* HTTP/1.0" 200 2326". A request for the profile of the user with identifier 5 would generate a log entry detailing which information was given to the web application (e.g. name, friends and profile picture), why it was handed out and so on. If an adversary could

correlate the log with the access log she could potentially link the log entry to a data subject (the one with identifier 5, shown in the profile at that URL) together with figuring out much of the contents of the entry. Furthermore, the adversary could profile the requests for log entries made by subjects, deducing when the subject in question is accessing his or her logs.

2.2.1 Shuffling to Remove Order

We focus our efforts on a proof-of-concept implementation of the privacy-preserving secure log that uses the HyperSQL DataBase (HSQLDB)[1]. Inspecting the *cached* and *text* table types in HSQLDB, they both leak the order in which entries are inserted into the database. For example, the text table type produces a Comma-Separated Values (CSV) file, where each line contains an entry. The last line contains the latest entry.

As part of our work on shuffling, we developed three versions of our "shuffler" [60]. We present the second version for sake of illustrating the approach, which we refer to as a *threaded shuffle*. The threaded shuffle has a probability of being triggered every time an entry is added to the log. Once triggered a new shuffle thread is spawned, leaving the main thread to continue its execution. The main thread and the shuffle thread share a coordinator object.

The Coordinator

The coordinator coordinates the access to the table containing all the log entries. It contains one flag and two locks:

- The active flag is set to true when a shuffle thread is running and false otherwise.
- The read lock is acquired to be allowed to read from the table in question while the active flag is set to true.
- The write lock is acquired to be allowed to write to the table in question while the active flag is set to true.

Modifications to Get and Add Methods

The methods for getting and adding an entry from the log has to be modified to take into account the coordinator object. When the coordinator is active the table

[1] http://hsqldb.org/, accessed 2015-03-20.

containing the log entries is not to be modified by anyone else than the shuffler thread. There is still a possibility that new entries are added to the log while the shuffler thread is running. For this reason, all attempts to add an entry to the log while the coordinator is active will result in the entry being placed in a temporary buffer. The get method, in the case of the coordinator being active, always acquires the read lock before retrieving an entry from the log or the buffer. The add method, when the coordinator is active, will acquire the write lock before writing the new entry to the buffer.

The Shuffler Thread

The shuffle is done in the following steps:

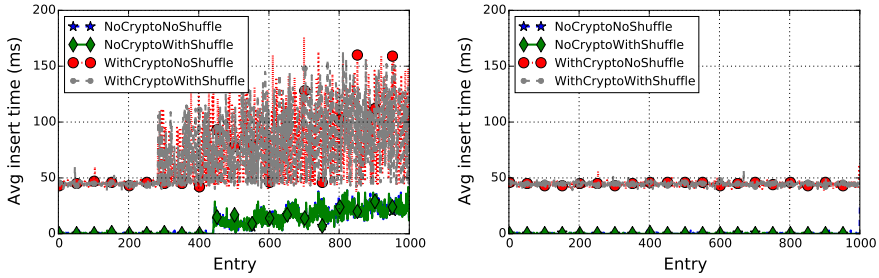
1. Create an empty table.
2. Get all the entries in the log, ordered by entry identifier and insert them into the internal table created in step 1.
3. Acquire the write lock.
4. Insert all the entries in the buffer (the entries that were inserted during the execution of the shuffler thread) into the internal table.
5. Acquire the read lock.
6. Disconnect the internal table from its data source and then disconnect the logs data source as well. Replace the logs data source with the internal tables data source.
7. Reconnect the log to its data source, which is now shuffled, and deactivate the coordinator (resulting in all locks being released).
8. Securely destroy[2] the old data source that was disconnected in step 6.

As a result of the process presented above, the data source ends up to a high degree being sorted based upon the entry identifier. The entry identifier is the output of a cryptographic hash function where the input is unknown to an adversary for entries generated prior to compromise. By inspecting the data source it is clear which entries have been shuffled. How many entries on average that are not sorted depend primarily on the shuffle rate, a configurable value.

[2]By securely destroy we mean removes all the data in the table and all possible files used internally by the database that could leak the order in which entries were inserted or any of the contents.

2.2.2 Performance Evaluation

We experimentally evaluate the overhead caused by the shuffler on the average insert time for new log entries. We focus on the average insert time since the shuffler is probabilistic, and alternate the size of the database since the shuffle acts on all log entries. Each experiment shows the average insertion time with and without cryptography and with and without the shuffler. "With cryptography" refers to the transformation to generate log entries for privacy-preserving secure logging, described in Section 2.1. There are two key configuration values of HSQLDB set for the experiments: the *scale factor* and the *write delay*. The scale factor refers to the size of the HSQLDB internal cache, and was at the time of the experiments defined as $3 \cdot 2^{\text{scale}}$ (presumably bytes). The write delay specifies the delay from changes to tables made in memory until the changes are synchronised with disk. We keep the write delay constant at 100 ms.



(a) HSQLDB scale factor 9.

(b) HSQLDB scale factor 11.

Figure 2.3 The overhead of the second version of the shuffler on average insert time from 10 runs for a new log entry. HSQLDB uses a write delay of 100ms. Figure 2.3a and Figure 2.3b use different HSQLDB scale factors.

Figure 2.3 shows the results of our experiments, run with two different scale factors. The experiments show no noticeable overhead at all for the shuffler, only the scale factor matters for HSQLDB. This is not all that surprising, since we have a dedicated thread for the shuffler, and writing into a buffer in HSQLDB or a buffer in the shuffler make little difference. With such a small database size (up to 1000 entries), the shuffler quickly gets the write lock (step 3), and then it is up to how quickly HSQLDB can store its entries. If the cache is full, then the time to synchronise with disk takes increasingly longer the more entries there are to write. The difference

between with and without cryptographic operations are due to the extra overhead in entry size produced by the cryptographic operations.

2.2.3 Key Contributions and Limitations

While the shuffler approach identifies and addresses a real problem, in general it has a number of limitations:

Reliability When the shuffler is active, entries are written into a buffer. During this time, entries in the buffer are presumably more vulnerable to data loss if the system goes offline.

Scalability The bigger the database, the more time the shuffler is active (time linear to the number of entries). While linear scaling is bad in and of itself, together with the reliability issue, it is a major limitation.

Forensics By creating and destroying new tables, with their own storage on disk, the shuffler approach is vulnerable to file-system and hardware forensics.

In general, the shuffler attempts to address a symptom (the database stored data in a way that leaks the order of insertion) instead of the cause (the design of the privacy-preserving secure logging scheme).

2.3 Logging of eGovernement Processes

Wouters *et al.* [122] presents a system focused on the citizen as a data subject, as part of the Flemish Interdisciplinary Institute for Broadband Technology (IBBT) Index project on intergovernmental data exchange. Citizens should be able to see how their personal data, stored in different governmental institutions, is used. Wouters *et al.* combined this idea with the idea of logging administrative processes. In the proposed system, the citizen is able to reconstruct a process, solely based on the events that are logged. This includes unfinished processes, which adds the ability to follow up on a process while it is executed. Next, we summarise their system with the goal of determining its contribution to defining the setting for transparency logging.

2.3.1 Setting

Each action or series of actions on the citizen's data can be seen as a process, often started by the citizen him- or herself. For instance, applying for a scholarship for a

child, retiring from one's work, changing address, etc. For each of these processes, data about the citizen has to be consulted, changed or added, and these actions can be logged. Moreover, the structure of these processes can be changed when laws are changed, so their structure can vary in time. Finally, logging (storing) and serving citizens' requests for logged events are services that are not really in the core business of governmental institutions, so we looked for a way to outsource them without violating the citizens' privacy, and having a minimal impact on existing processes.

The main assumption in the attacker model is that an entity who executes (part of) a process can be trusted to some extent, in a certain time-frame. If this is not the case, there exists no trustworthy data to be logged in the first place. In the attacker model, which was not formalised, they distinguish the following entities:

Outsiders These have only access to the network.

Data logging insiders These attackers have access to the logging services of one or more log servers.

Data processor insiders These have all available access rights to all data that flows through one or more data processors.

All these entities are global and active. This implies that they can eavesdrop on all communications at all times, and can change data communications at will, and have administrator privileges. The inside attackers in this model will try to change or delete existing log records, or add new log entries in between them. They can trivially tamper with all newly added log entries. Outsiders can be present during the entire life cycle of a process and its logging. The focus of the protection mechanisms in the model is on the possibility to reconstruct a process and possibly link it to a data subject, based on the possibly distributed database of logged events only. Therefore, they also assume that outsiders have access to all logged events, through the same means as benign data subjects, querying the log server to reconstruct the status of their process.

Within this attacker model, the system should meet the following requirements:

- With the aid of the data on the log servers, it should be possible for the data subject to determine which institutions have completed which parts of his or her process, and to verify if the institutions are not stalling the process.

- Institutions are free to choose between setting up their own log server or using a third party service for logging. A third party service can be a completely

independent entity—logging completely outsourced, out of control of the institution—or a semi-trusted entity within the institution's environment. For instance, institutions within different governmental services might all log to one central logging service, maintained by the government. In each case, they assume that the logged data is publicly accessible.

- The data subject of the process is able to verify the status of his or her process but he or she can also delegate this to a third party.
- If log servers collude, they should not be able to link their logging events, when only the logged data is available.

In the following sections, we describe the core functionality of their system. More detailed information and a description of the actual implementation of the system can be found in [70].

2.3.2 Building the Trail

In their system, a process, started by a citizen, leaves a trail of log events across a network of logging servers. The links connecting the log events in different logging servers are protected in such a way that the complete trail can only be reconstructed by the citizen or by someone he or she delegates. The handling of the process itself, and how it is passed between institutions, is not described. Confidentiality of what is being logged is an essential part of the logging process.

We clarify the construction of the logging trail by an example with four different parties MR, A, B and C. MR (the Mandate Repository) acts as a front-end that the citizen will use to start, organise and consult his or her processes. It also contains cryptographic key material for processes of that user, and can serve as a mandate broker to grant access to a certain set of logs of a process, to civil servants and other parties. In the following, we describe the part of the process in which logging events are entered into the log servers (see also Figure 2.4).

The process is started for the citizen at MR. It is logged as follows:

1. MR generates a process identifier μ, which is recorded for (and maybe by) the citizen. This process identifier must be chosen at random. It also generates a key-pair (PuK, PrK), signed by MR, to secure the logging events for this process, also to be kept by the citizen. This key-pair must be process-specific, such that it can be disclosed to civil servants in case they have to check on the status of

the process[3]. Furthermore, it calculates $\mu' = H(\mu)$ and $\mu'' = H(\mu')$, where $H(\cdot)$ is a collision-resistant hash function. It also determines a random 'pointer' p_m, which connects μ'' to the logged data of the next step in the process. μ'' is the identifier of the process within the log server L_{MR}.

2. (μ'', p_m) is logged on the log server L_{MR} of MR[4].

3. MR generates $m = H(\mu'|p_m)$ and sends this value, together with the process, PuK and the URI of the log server that MR is using, to A, the first party in the process chain. This value m will enable A to provide a user with a link to the log server of A.

4. When A receives m, it generates its own random internal process identifier α, together with $\alpha' = H(\alpha)$ and $\alpha'' = H(\alpha')$.

5. Now, A generates a log event in L_{MR} that links the previous step in the process to the current one. It sends a pair (m', m_A), where

$$m' = H(m), \quad m_A = E_{\mathrm{PuK}}(\alpha'|m|\mathrm{URI}_{L_A})$$

and $E_{\mathrm{PuK}}(x)$ is a KEM/DEM (Key Encapsulation Mechanism -Data Encapsulation Mechanism) hybrid cipher (see [63, 108]). Note that (m', m_A) cannot be linked to (μ'', p_m) if μ' is unknown, which means that even the log server L_{MR} cannot see which process went to another log server[5]. The URI indicates the location of the log server on which A is going to log the events for process α.

6. A performs the steps for its part of the process, and doing so, it logs events (milestones), related to these steps and lists them under α''. If the logged data contains sensitive information, it can be encrypted under the public key of the process.

[3] Note that it may be possible that the administrations have their own process status checking tools. This proposal tries to enable that functionality towards the citizen. We believe that there are good reasons to keep status checking by the citizen separated from the processing servers of the administrations (in fact, from any server of the administration).

[4] In Figure 2.4, this action is labeled as 2) in L_{MR}. Label 1), used in the other log servers, is reserved for log items that contain information about the actual process.

[5] Note that they do not consider traffic analysis here. If a log server has an extremely low load, he will be able to track where a process went, just by timing logged events.

7. In the depicted scenario, for completing the process, A calls B and C. For each of these branches, A generates a random 'pointer' value: p_{a_1} for B and p_{a_2} for C, calculates $a_1 = H(\alpha'|p_{a_1})$ and $a_2 = H(\alpha'|p_{a_2})$, and sends these values to B resp. C. To L_A, it sends the pairs (α'', p_{a_1}) and (α'', p_{a_2}).

8. When C and B receive their calls from A, they behave in the same way as A did, i.e., they generate their own process identifier, and send the necessary linking information to the log server of A.

We define μ, α and β to be the internal process identifier, while μ', α', β' and μ'', α'', β'' are the first and second order local process identifier. Only the second order identifier is stored at and visible to the log server, while the first order process identifier is only known by the institution that generates it and the citizen who owns the process.

2.3.3 Reconstructing the Trail

When a citizen wants to check the status of his or her process, he or she will only have to query the log servers.

- The citizen has μ, so he or she can construct μ' and μ''.

- He or she queries L_{MR} for μ'' and retrieves p_m.

- The citizen can then compute $m = H(\mu'|p_m)$ and $m' = H(m)$, and look up entry (m', m_A) in L_{MR} and decrypt m_A, such that α' and URI_{L_A} are known. Because m was only known by MR and A, the log entry (m', m_A) that reveals α' can only be retrieved by the citizen (and MR and A).

- URI_A can now be queried for all entries listed under $\alpha'' = H(\alpha')$. The citizen retrieves the (encrypted) log data about actions that A performed, and identifiers p_{a_1} and p_{a_2}.

- Identifiers p_{a_1} and p_{a_2} give access to a_{1B} and a_{2C}, which can be decrypted to follow the link to L_B and L_C.

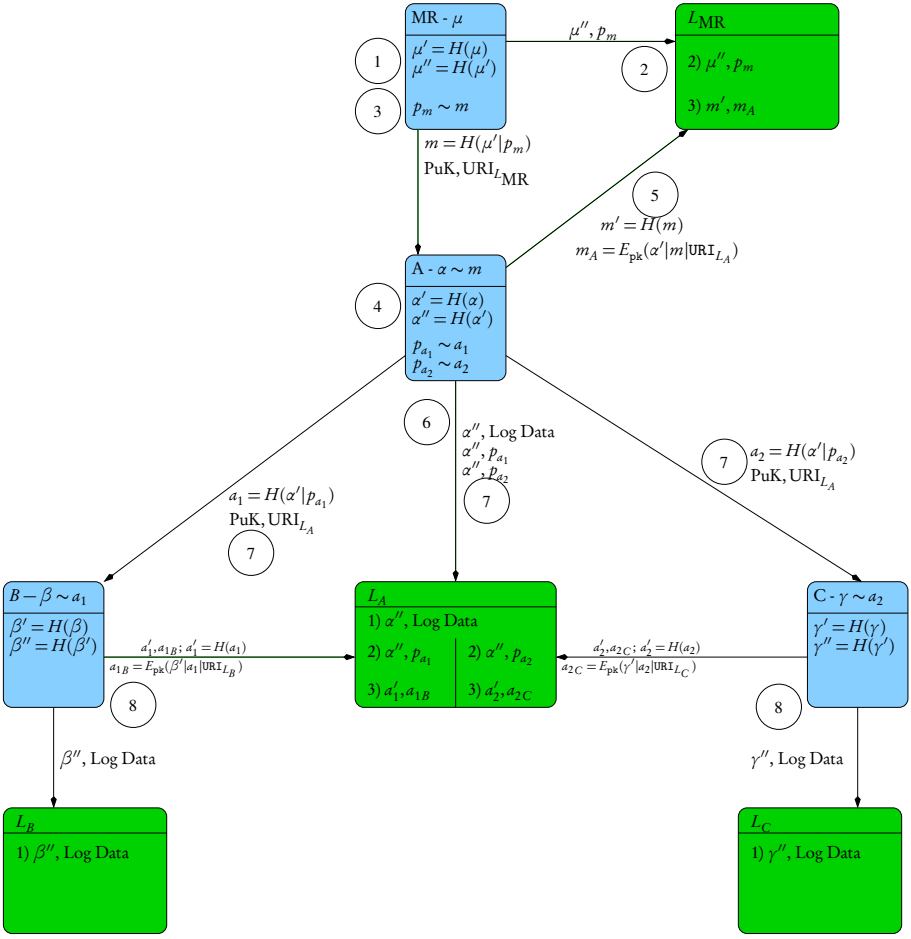


Figure 2.4 Logging a process in the model of Wouters *et al.* [122] .

In Summary,

$$
\begin{aligned}
\mu \longrightarrow \mu' \longrightarrow \mu'' &\implies L_{\mathrm{MR}} : \mu'', p_m \\
\mu' | p_m \longrightarrow m \longrightarrow m' &\implies L_{\mathrm{MR}} : m', m_A(\alpha', L_A) \\
\alpha' \longrightarrow \alpha'' &\implies L_{\mathrm{A}} : \alpha'', \text{Log Data}, p_{\alpha_1}, p_{\alpha_2}, \\
\alpha' | p_{\alpha_1} \longrightarrow a_1 \longrightarrow a_1' &\implies L_{\mathrm{A}} : a_1', a_{1_B}(\beta', L_B) \\
\alpha' | p_{\alpha_2} \longrightarrow a_2 \longrightarrow a_2' &\implies L_{\mathrm{A}} : a_2', a_{2_C}(\gamma', L_C) \\
\beta' \longrightarrow \beta'' &\implies L_{\mathrm{B}} : \beta'', \text{Log Data} \\
\gamma' \longrightarrow \gamma'' &\implies L_{\mathrm{C}} : \gamma'', \text{Log Data}
\end{aligned}
$$

2.3.4 Auditable Logging

With the construction above, citizens' data remains confidential, and reconstructing process information from the logged events, without direct access to the first-order local process identifiers, is computationally hard. However, it is still possible for a malicious log server or an attacker to delete or change logged events, or to confuse (random) citizens by entering dummy events under existing second order local process identifiers. These attacks are aimed at the availability and the authenticity of the log. To preclude these attacks, a signature/time-stamping scheme can be implemented, depicted in Figure 2.5. This scheme ensures the following:

Institution towards the log server and citizen Periodically, the institution will send a signature to the log server. This signature will cover the logged events sent to the log server since the previous signature. We will refer to this time interval as a round. The institution will have to keep a history of the events of the current round, for every log server that it is using. We formalise this as follows:
The institution A keeps a record $\mathscr{L}_{L_A}^{(r_i)}$ of the logged – but yet to be signed – events $l_1^{(r_i)}, \ldots, l_j^{(r_i)}$ that were sent to each log server L_A. The number r_i indicates the current round and j the number of the logged events in the round. When a round finishes, the institution will calculate $S_A(H(\mathscr{L}_{L_A}^{(r_i)})|L_A)$ and send this to L_A. L_A verifies the signature, stores it, and marks the logged events in its database as verified. If the verification fails, L_A notifies A, and marks the events in its database as untrusted. Using these signatures, log servers can show to external parties (citizens, auditors) that the events they store in their log are genuine and originating from a certain institution.

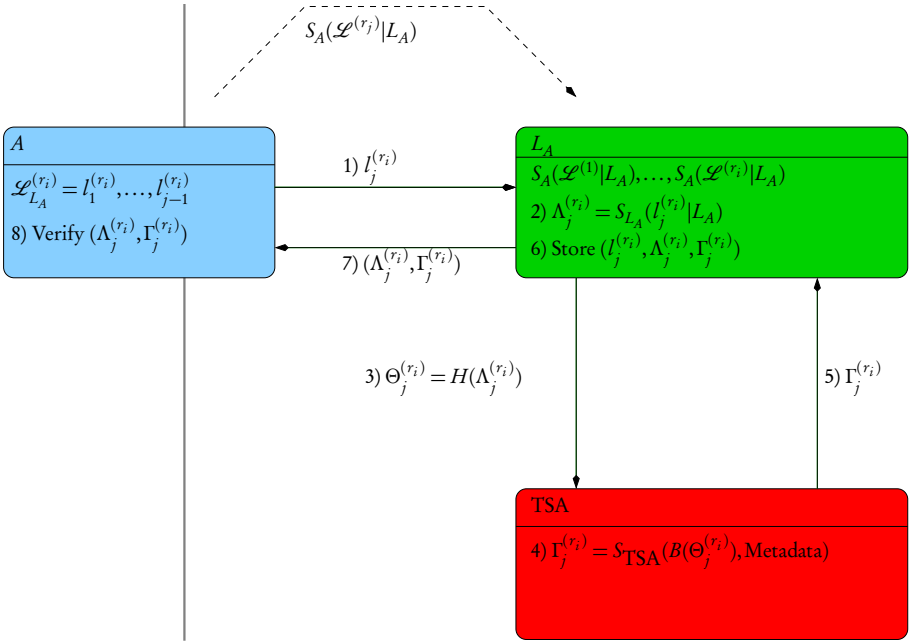


Figure 2.5 Making a log server auditable.

Log server towards institution The institution must be able to prove to the citizen that it has logged the substeps of the process for which the citizen is building a status report, and that it performed its duties towards that citizen. This is necessary if a process gets stuck in a certain entity (institution), or if the reconstruction of a process log fails. If L_A claims that a certain log entry that A knows it has logged, does not exist, A should be able force the log server to produce the entry anyway, or to admit that he did not follow the service agreement with A. In other words, L_A must commit to have logged the log entries, submitted by A, in such a way that A can reuse this commitment as a proof later on. This enforces the service agreement that an institution has with a log server. It is enforced by having the log server sign each submitted log entry: $\Lambda_j^{(r_i)} = S_{L_A}(l_j^{(r_i)}|L_A)$. This signature will be sent to the submitting institution.

Timeliness and completeness Even if a log server is responding with the signatures mentioned above, disputes about timeliness can still arise. For instance, a log server might follow the protocol towards the institution, but might have an under-performing service towards citizens, regularly replying to requests with an 'entry does not exist' message, later on claiming that the entry did not exist yet, at the time of querying. In addition, the above mechanism does not allow an institute to ask for a full reproduction of its logged events at a certain log server, without actually reproducing all signatures from the log server. Therefore, a time-stamping authority (TSA), producing linked time-stamps, will be used: when the log server receives an event $l_j^{(r_i)}$ to be logged, it will first compute the signature $\Lambda_j^{(r_i)} = S_{L_A}(l_j^{(r_i)}|L_A)$. Then, it forwards the message digest $H(\Lambda_j^{(r_i)})$ of the signed log event to the TSA, to link it to the previously time-stamped events $\Lambda_l^{(r_k)}$. The TSA computes linking information $B(H(\Lambda_i))$ containing the necessary information to verify the link, adds some metadata to it (serial number, time value, etc.), signs it and returns this as the result to the log server, that forwards it to the institution. At reception, the institution should always verify the signature(s) and the time-stamp. Then, if the institution ever gets challenged by citizen, it can ask for a complete replication of all the logged events, including the signature of the log server, based on the most recent time-stamp it possesses. This will expose all the logged events because of the linear linking in the time-stamp.

2.3.5 Key Contributions and Limitations

The work on logging of eGovernment processes makes a number of key contributions for privacy-preserving transparency logging:

Dynamic distributed settings Wouters *et al.* identify the need for a dynamic distributed log system. By spreading "linking information" to log servers, the citizen can reconstruct his or her log trail by following the links without a-priori knowledge of the structure of the process.

Dedicated log servers Storing and serving log entries are not a core task of data processors, so being able to outsource such activities are compelling.

Auditing Beyond authenticity needs of the logs, auditing becomes an issue with multiple players with their own interests of showing that they did their part of logging a process. A log server may misbehave towards citizens by claiming

that log entries do not exists, and proving timeliness with the help of a TSA is a possible remedy.

However, it has some limitations:

Log items within one log server Several logged items of one process on the same log server are trivially linkable by anybody who has access to the log database, due to the fact that they are logged under the same process identifier.

Lack of integrity protection There is no integrity protection beyond what is offered by auditing.

No formal attacker or adversary model There is no formally defined adversary or attacker model.

Next, we define the setting of transparency logging.

2.4 Defining the Setting of Transparency Logging

The primary players are one or more *controllers*, *users*, and *servers*. A controller controls all data stored in a service provided by the controller. We opt for the term controller to emphasize the actual control, in the technical sense, over the data in the service provided by the controller. Note that this is in stark difference to the concept of a "data controller" from the EU Data Protection Directive 95/46/EC, where the "data controller" in general at best can exercise some legal control over data not on its own systems [44]. The data at the controller is personal data about the users of the service. The controller uses a server to log all processing it does on the users' personal data, i.e., to perform transparency logging. Servers are dedicated to transparency logging, while the controllers' primary purpose is to provide a service to the users. Users retrieve their logs from the servers. Note that controllers do not directly send the logs to users because users cannot be assumed to have high availability. Figure 2.6 illustrates the primary players and their interactions. Dashed lines indicate interactions due to transparency logging, while thick lines indicate a data flow due to the service provided by the controller.

Services are often composed of other services, similar to how processes and data processing in general are often distributed. This means that there may be multiple controllers connected in any directed graph to facilitate a service, or as part of multiple services sharing personal data. Figure 2.7 shows an example of a distributed setting

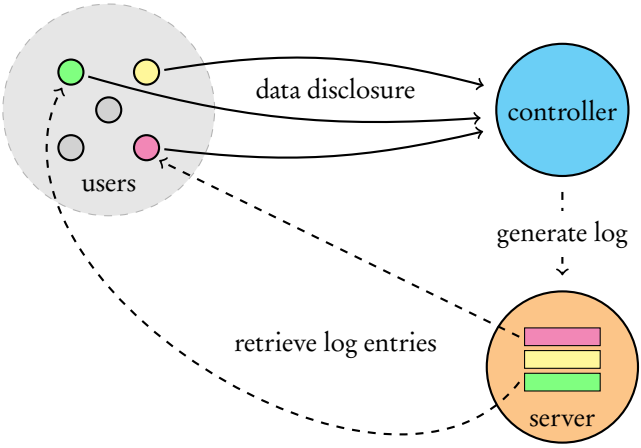


Figure 2.6 The primary players in transparency logging and their interactions. The players are users, a controller, and a server. In this example, the green, yellow, and pink users disclose personal data to the service of the controller. The controller generates the log at the server, and two users (pink and green) retrieve their log entries from the server.

with four controllers (Bob, Carlos, Dave, and Frank) and three active users (green, yellow, and pink dots). The users use the service provided by Bob, Bob uses the service provided by Carlos and Carlos in turn uses the services of Dave and Frank to provide his service. Therefore, when the users disclose data to Bob, the data will be shared by Bob with Carlos, who in turn shares it with Dave and Frank. Transparency logging should support such distributed settings in a *dynamic* way, i.e., the structure of how the controllers depend on each other should be able to change and not set at the time of data disclosure.

The last piece missing in Figure 2.7 is servers. Just like controllers can be dynamically interconnected to facilitate a service, the selection of servers for transparency logging could be dynamic. Controllers could use their own servers, or share servers with controllers. Figure 2.8 introduces servers to the example setting of Figure 2.7. In Figure 2.8, Bob and Carlos share one server, while Dave and Frank share another server. The pink user retrieves his or her logs from the servers.

Beyond the primary players, there are some other relevant third-parties as well. Primarily, the adversary is not yet defined. The adversary, and how it is defined, evolves throughout this dissertation. Secondly, there are other (presumably trusted) parties, like an auditor, a judge, time-stamping authorities and monitors. An auditor

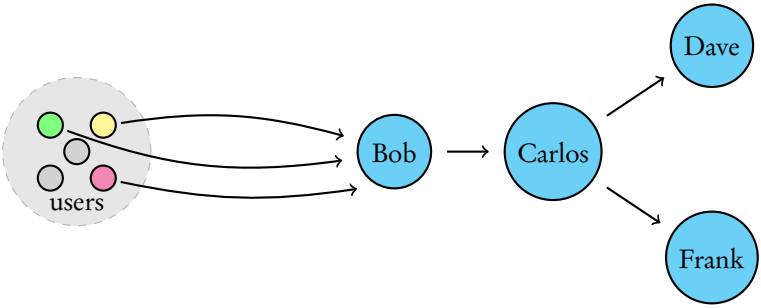


Figure 2.7 The involvement of controllers should be dynamic. An example distributed setting with three active users and four controllers. Users (green, yellow, and pink) disclose personal data to the controller (blue) Bob. Bob shares some of the disclosed data with controller Carlos, who in turn shares it with controllers Dave and Frank.

is trusted to audit some part of a transparency logging system. A fair judge will, e.g., punish a misbehaving controller when presented with proof. If the judge is not fair, any proof is moot. Time-stamping authorities are assumed to faithfully attest the time of some data. Finally, monitors, like auditors, are assumed to monitor some parts of a system. We will return to these players later.

2.5 Requirements

Next, we derive a number of high-level requirements for privacy-preserving transparency logging based on our setting and the early work presented in Sections 2.1–2.3. The requirements are divided into functional, verifiable integrity, privacy, and auditability and accountability requirements.

2.5.1 Functional Requirements

We define a process as all the data processing performed by controllers on personal data disclosed by a user to a service where the user is the data subject. For each new such process, a different log trail will be generated. The log trail consists of log entries, possibly stored at different servers (by the controllers), and can be distributed over several servers. The primary purpose of the log trail is to enable a user to reconstruct the actions belonging to that process. First, this means that he or she can identify the log entries that relate to a certain process he or she owns. Subsequently, he or she should be able to rebuild the structure of the log trail, which will give, assisted by

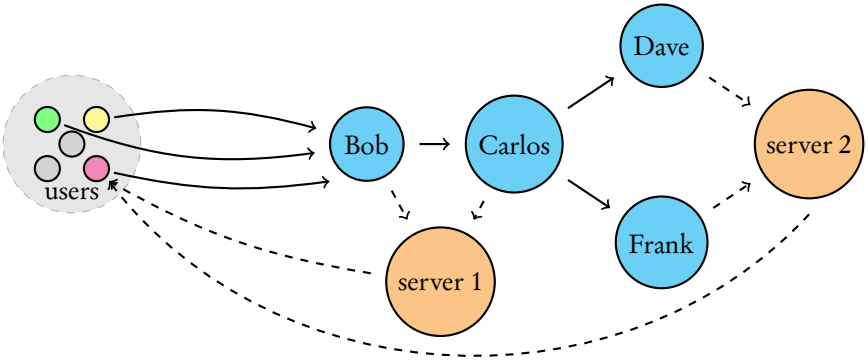


Figure 2.8 The involvement of servers should be dynamic, just like controllers. The final example of a setting for transparency logging with three active users, four controllers, and two servers. In this example, the pink user can retrieve his or her logs from the two servers, where the logs have been generated by the controllers.

the content of the logged entries, a representation of the process itself. This ability should be limited to the user, leading to the following functional requirements:

R1 A user should be able to identify and retrieve all log entries logged for him or her.

R2 A user should be able to structure the logged entries in a log trail, and read their (plaintext) content.

R3 The abilities mentioned in R1 and R2 are restricted to only the user for whom the log entries have been logged.

2.5.2 Verifiable Integrity

The log at a server, consisting of log entries, should be protected against tampering. This means that entries cannot be modified or deleted without detection. We note that that accurate information is a privacy principle in the EU Data Protection Directive 95/46/EC, and that tampering especially by the controller after the fact to attempt to cover its tracks is a realistic threat. Bellare and Yee [18] refer to this property as *forward integrity* in the closely related secure logging setting. Note that this property implies an adversary in the forward security model, i.e., the controller and potentially server is assumed initially trusted and at some point in time turns into an active adversary. The goal is to protect entries generated prior to compromise.

In our system we assume three possible classes of verifiers, leading to the following requirements with respect to verifiable integrity:

R4 The user should be able to verify the integrity of the log entries related to his or her processes.

R5 The controller should be able to verify the integrity of the log entries it was part of generating.

R6 For enabling R5, a controller should be able to easily identify and retrieve the log entries that it generated.

R7 An auditor should be able to verify the integrity of a complete log kept by a server, assisted by the controllers who generated entries in the log.

To summarise, entities should be, independently of each other, able to verify the integrity of those log entries that are related to them (users) or that they themselves generated (controllers). Third parties, like auditors, may also be interested in verifying the integrity of a log.

2.5.3 Privacy

The content of log entries should remain confidential since they are personal data. This is one of the often-cited requirements in papers on secure logging, and is captured by the functional requirement R3 above. For privacy-preserving transparency logging we have multiple users and can go one step further: log entries should be *unlinkable*. Apart from protecting which information was logged about a certain user, we also want to hide the mere fact that something was logged for an identified user, or that a certain subset of the log entries relate to the same user. These requirements should hold for entities that are not related to the log entries: because of requirement R5, controllers are able to identify the log entries they generated, while users are able to identify those entries that relate to them, due to requirement R1.

Given a subset W of all log entries, the following requirements should hold for entities E that are not related to and who did not produce any log entry in W. Given the information in W, for E:

R8 There should be unlinkability between different log entries. It should be computationally infeasible to tell if two log entries $e_1, e_2 \in W$ are related or not. A possible relation can be a temporal ordering, determining whether or not e_1 was logged before e_2.

R9 There should be unlinkability between log entries and user identifiers. Given a user identifier of a user $U_1 \notin E$, used to generate at least one log entry, it should be computationally infeasible to link it with any log entry committed to the log. Similarly, if given a log entry in W it should be computationally infeasible to determine which user identifier it belongs to.

The log entries in W will be generated using specific metadata in the form of state. Each time a log entry is generated, the state will be updated, like the ratchet in Section 2.1. The updated state should not reveal any useful information about previous log entries in W. Given access to state after generating W:

R10 There should be unlinkability between state and any log entry in W. This can be seen as forward unlinkability. In other words, when an attacker gets access to state any entries already generated with state should be unlinkable to the state itself.

R11 There should be unlinkability between different user identifiers that are part of state. If a process is taking place across multiple controllers and servers it should not be possible to determine the path of the process by inspecting state.

These requirements should hold for user identifiers and log entries across different controllers and servers.

2.5.4 Auditability and Accountability

Transparency logging should be auditable by an auditor, as noted in Section 2.3. Moreover, the controllers and servers should have the ability to hold their counterparts accountable. Controllers and servers are operating under a certain contract: the server plays a role in transparency logging for the users of the service provided by the controller. We explicitly set the following requirements:

R12 Towards a controller, a server should be able to show that the log entries in the log could not have been back-dated, deleted or altered after a certain point in time. It should also be able to show that only data sent by the controller to be logged have been turned into log entries in the log.

R13 Towards a user, a server should be able to show that the data in log entries for the user were approved by the controller, and that these entries could not have been back-dated, deleted or altered after a certain point in time.

R14 Towards an auditor, a server should be able to show that its log consists of log entries approved by an identifiable set of controllers, and that the log could not have been tampered with.

R15 Towards an auditor, a controller should be able to show that a server accepted and approved a given subset of log entries that it was part of generating at the log server.

R14 and R15 concern a controller and a server being able to keep each other accountable towards an auditor with regard to all log entries in a log. In other words, they can demonstrate to an auditor that the other party agreed to the current state of the log.

2.6 Moving Forward

Early work by Hedbom *et al.* [62] and Wouters *et al.* [122] identified key characteristics of the transparency logging setting. Data processing by controllers is distributed and dynamic. Key privacy issues are related to linking log entries and user identifiers. Dedicated servers for storing logs enable controllers to focus on their core activity of providing a service while offloading parts of transparency logging to servers. Moving forward, we focus on formally defining and proving key security and privacy properties for distributed privacy-preserving log trails that enable transparency logging. To better illustrate privacy issues in the setting, we start the next chapter with an example of using transparency logging for making access to electronic health records transparent to patients.

Chapter 3

Distributed Privacy-Preserving Transparency Logging

> For what is algorithm but ideology in executable form?
>
> *Lisa Nakamura*

This chapter presents a cryptographic scheme for distributed transparency logging of data processing while preserving the privacy of users. Our scheme can handle arbitrary processes while offloading storage and interactions with users to dedicated log servers. Compared to earlier related work, our scheme is the first to define and formally prove several privacy properties in this setting.

The structure of this chapter is as follows. First, Section 3.1 shows how transparency logging can be used in a healthcare setting and related privacy issues that our scheme addresses. Section 3.2 discusses related work with a focus on secure logging. Section 3.3 presents our general model and definitions. This is followed in Section 3.4 by a thorough explanation of our scheme. Section 3.5 evaluates the scheme's security and privacy properties. We present a performance evaluation of our prototype implementation in Section 3.6. Finally, Section 3.7 provides concluding remarks.

3.1 Introduction

Transparency of data processing is often a requirement for compliance to legislation and/or business requirements; e.g., keeping access logs to electronic health records (EHR), generating bookkeeping records or logging data to black boxes. Furthermore, transparency is recognised as a key privacy principle, e.g., in the EU Data Protection Directive (DPD) 95/46/EC Articles 7, 10, and 11; and in the Swedish Patient Data Act ("Patientdatalagen") SFS (2008:355) that states that patients have the right to see who has accessed their EHR. In general, increased transparency of data processing may increase the end-users' trust in the controller,especially if the data processing is distributed as in cloud computing [67]. Other applications can be found in eGovernment services, where enhanced transparency towards citizens is a key element [114].

Beyond technical considerations of *what* to make transparent to *whom* to accurately represent data processing, there are a number of social and economic issues that need to be taken into account when determining what an adequate level of transparency is. As recognised in recital 41 of the EU DPD, too detailed descriptions of data processing may reveal business-sensitive information of the controllers, such as trade secrets. Furthermore, employees of a controller may experience the requirement of transparency as a breach of their own privacy [98]. Our work is agnostic to what information is made transparent. We focus on ensuring that only the user whose personal data is being processed can read the logged information. This conservative approach ensures the end-users' privacy. In general, sharing information is easier than removing access to information that has already been provided to a party. Our approach has the added benefit of reducing the negative effects of transparency on the employees of the controller, since their actions are only made transparent towards the user whose data they are processing. For instance, the logs cannot be misused to monitor the employees' performance.

Information about how data will be processed by controllers is usually represented by a *privacy policy* that states what data is requested for which purpose, whether the data will be forwarded to third-parties, how long the data will be retained, and so on. A privacy policy informs the potential user *before* any data is disclosed, so that a user can give *informed consent* to the data processing by the controller. Our approach enables controllers to inform users about the actual data processing that occurs *after* users have disclosed data, taking the privacy of users into account. Conceptually,

access to this information enables a user to *verify* that the actions of the controller are in line with the privacy policy[1].

Our cryptographic scheme allows a controller to use a set of servers, creating log entries that describe how the user's data has been processed. Users can then, at their own discretion, query these logs to learn of the data processing performed on their disclosed data. Figure 3.1 illustrates this setting between a user Alice and controller Bob. Alice discloses some data to Bob under a privacy policy. While Bob is processing Alice's data, Bob generates log entries at a log server that records the processing. Alice can later retrieve the log entries that contain the description of data processing and verify if Bob's actions are in line with his privacy policy.

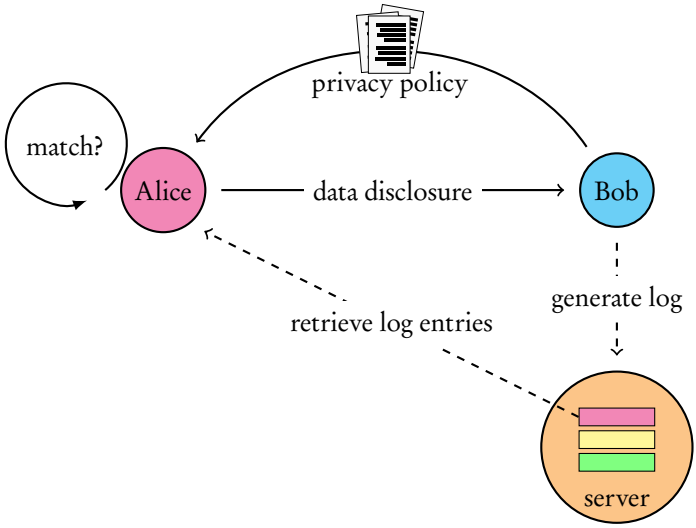


Figure 3.1 High-level overview of how transparency logging can be used.

The simple setting illustrated in Figure 3.1 is missing one important aspect of data processing: data processing is often distributed. Data concerning users may be shared by the controller, or other controllers may be used by the initial controller. We denote the actions, performed by a set of controllers, for a particular task, a *process*. Figure 3.2 illustrates a setting for a process (solid lines) where Alice discloses data to Bob, the initial controller. Bob then shares (part of) Alice's data with the downstream

[1]Assuming that (1) the semantics of the privacy policy and information provided to the user allow reasonable comparison, and (2) that the information provided by the controller is *complete* (see Section 3.3.1).

controller Carlos, who in turn shares (part of) Alice's data with controllers Dave and Frank. While controllers Bob, Carlos, Dave, and Frank process Alice's data, all of them continuously log descriptions of their processing (dashed lines) to their, potentially different, log servers. Alice can later reconstruct the *log trail* of the data processing on her data.

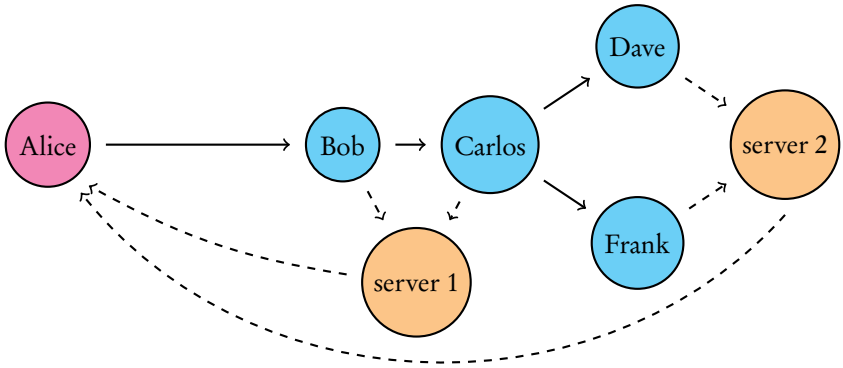


Figure 3.2 Data processing is often distributed, where data disclosed by a user *Alice* may be shared by the initial controller *Bob* with other controllers *Carlos*, *Dave*, and *Frank*. Downstream data processing of Alice's data should also be logged.

In the case of an EHR system, in Figure 3.2, Alice could be the patient and Bob, Carlos, Dave, and Frank medical journal systems at one or more medical institutions. All access by medical staff to Alice's medical records will be logged such that Alice can later see who has read her medical records.

Because log entries contain descriptions of data processing on personal data, the log entries themselves also contain personal data. Therefore, when implementing such a log system, there are several privacy and security issues that need to be addressed. Returning to the EHR example, knowing who has accessed a person's medical records is sensitive information. In fact, even knowing that a person has been treated at a particular medical institution may be sensitive. In other words, not only the content of the log entries but also the fact that log entries exist for a certain individual is sensitive information. Furthermore, imagine the case of medical personnel illegitimately accessing EHRs. In such a case, the offenders are also likely to attempt to cover the traces of their actions, for example by deleting the generated log entries. Therefore, alterations to log entries need to be detectable once they have been constructed. Similar arguments can be made for the case of business-sensitive information being logged for a company.

3.2 Related Work

Sackmann *et al.* [103] presented the earliest work on providing transparency of data processing by using cryptographic systems from the secure logging area. Trusted auditors use secure logs of data processing as so called "privacy evidence" to compare the actual processing with the processing stated in a privacy policy that users have consented to. Wouters *et al.* [122] and Hedbom *et al.* [62] tackle privacy issues in a setting with one key difference from the work of Sackmann *et al.*: users take on the primary role of auditors of the logged data that relates to them, arguably removing the need for trusted auditors. This change of setting is the primary source of potential threats to the privacy of users (ignoring the content of log entries), since log entries now relate to different users who are actively participating in the scheme. These threats are primarily due to the *linkability* of entries to users and user identifiers between logs. Wouters *et al.* address the linkability issue between logs on different log servers in a distributed setting, while Hedbom *et al.* address linkability between log entries in one log. Chapter 2 summarised these schemes, and Section 1.5 provided a summary overview. The scheme we propose in this chapter addresses several issues in prior work, such as truncation attacks (described later) and partial linkability of log entries. Furthermore, we propose generalised notions of security and privacy in this setting and prove that our scheme fulfills these notions.

Note that our work, like [62, 103], builds upon concepts from the secure logging system by Schneier and Kelsey [104]. A thorough review of related work in the secure logging area follows. We focus the review on the following four features:

Integrity of the log Most papers on secure logging start off with solving this problem. The main concern is detection of log compromise, and in some cases, being able to detect which logs were left uncompromised.

Confidentiality of the log In some cases, the log needs to be kept confidential because it contains valuable (business or sensitive personal) information. Other papers also mention the privacy of the subjects to which the log entries refer.

Access control and auditability Apart from classical access control mechanisms, encryption is used to prevent access to cleartext data. Recovery of the decryption key is sometimes combined with access control and searchability.

Searchability When log entries are encrypted, searching in them becomes virtually impossible. This can be resolved by other means such as adding encrypted keywords.

3.2.1 Early Work on Secure Logs

Several papers refer in their related work section to the work of Bellare and Yee from 1997 [18] and Schneier and Kelsey from 1998 [104] for the first results in secure logging and more in particular the use of hash chains to protect the integrity of a log file. However, in 1995 Futoransky and Kargieman [52] wrote about a basic version of a hash chain algorithm to protect the integrity of a log file and to encrypt the entries. They refined their work on PEO ("Primer estado aculto", meaning "hidden first state") and VCR ("Vector de claves remontante" meaning "remounting Key Vector") in a later paper [53]. They also implemented their work as a secure UNIX syslog daemon, and a secure event logger for Windows NT. The essence of their protocol is a hash chain K_i that depends on the submitted log entries D_i:

$$K_i = \texttt{Hash}(K_{i-1}, D_i) \qquad C_i = \texttt{Enc}_{K_{i-1}}(D_i).$$

The log then stores C_i and overwrites K_{i-1} with K_i. In this simple version, a truncation attack (explained in Section 3.2.3) cannot be detected by the secure log system, unless intermediate log entries are submitted to an auditing server.

In 1997, Bellare and Yee [18] introduced the notion of epochs (or time intervals) to achieve forward integrity (FI) of a log. Taking a more formal approach, they also provided a definition for FI secure log systems, and deletion detecting-FI (DD-FI) secure log systems. In their solution, a log entry is authenticated with a MAC algorithm under a key that is unique for each epoch. The key evolves by applying a pseudo-random function. Even if a system is compromised in a certain epoch, all entries logged in previous epochs are secure against modification. To be able to detect deletion of log entries (DD-FI), sequence numbers and epoch change markers are added. For a submitted log entry m_i the log stores $(m_i, \text{FIMAC}_j(m_i))$, where $\text{FIMAC}_j = \text{MAC}_{k_j}$ is a MAC algorithm with a key k_j generated by an appropriate pseudo-random function (*prf*), evaluated in a chain:

$$k_j = prf_{s_{j-1}}(0) \qquad s_j = prf_{s_{j-1}}(1).$$

The disadvantage of their solution, being based on symmetric primitives, is that the initial keys have to be shared with the verifier. Bellare and Yee actually prove that the FI of their scheme can be reduced to the security of the underlying MAC algorithm and the *prf*. They also implemented the scheme, with HMAC-MD5 as the MAC algorithm and the IDEA block cipher as the pseudo-random function.

In [104, 106], Schneier and Kelsey use hash chains combined with evolving keys, similar to the approach of Futoransky and Kargieman [52]. In a more formal approach, they define an untrusted log server $\mathcal{U}$, a trusted party $\mathcal{T}$ and a set of (moderately-trusted) verifiers $\mathcal{V}$ that can have access to certain parts of the log. The log entries of an untrusted log server are periodically synchronised with the trusted server $\mathcal{T}$. Verifiers $\mathcal{V}$ can query the trusted server $\mathcal{T}$ to get access to log entries, even if they reside only on $\mathcal{U}$. Given a data item D_i to be logged, and an evolving key A_i, the log file entries L_i look as follows:

$$
\begin{aligned}
L_i &= (W_i, \mathtt{Enc}_{K_i}(D_i), Y_i, Z_i), \text{where} \\
K_i &= \mathtt{Hash}(\text{“Encryption Key"}, W_i, A_i), \\
Y_i &= \mathtt{Hash}(Y_{i-1}, \mathtt{Enc}_{K_i}(D_i), W_i), \\
Z_i &= \mathtt{MAC}_{A_i}(Y_i).
\end{aligned}
$$

The key A_i evolves by hashing it: $A_{i+1} = H(\text{“Increment Hash"}, A_i)$, while the entry W_i is a permission mask to determine access control by the external verifiers $\mathcal{V}$. New values for A_i and K_i irretrievably overwrite the old values. The initial value A_0 of the key A_i is shared between the untrusted log server $\mathcal{U}$ the trusted party $\mathcal{T}$. The integrity of the log is protected by the two elements Y_i and Z_i. Y_i establishes a hash chain that depends only on values that are in the log, and can therefore be used by verifiers $\mathcal{V}$ to check parts of the log. The element Z_i allows a trusted server $\mathcal{T}$ to verify that a request from $\mathcal{V}$ for access to unsynchronised log entries on $\mathcal{U}$ is in fact genuine; because $\mathcal{T}$ has the initial value A_0, it can generate any A_i and verify the MAC on Y_i. When a verifier $\mathcal{V}$ wants to have access to the log, he or she will identify to $\mathcal{T}$, also passing on his or her permission mask, which will reveal the necessary keys K_j to decrypt log entries. Apart from the log entry structure, Schneier and Kelsey describe a complete protocol on how a log database is initialised, closed and accessed. To establish the initial authentication key A_0 between $\mathcal{U}$ and $\mathcal{T}$, they use a public key infrastructure to sign and encrypt messages during initialisation. Schneier and Kelsey also discuss extending their protocol over a network of peers to enable cross-linking of log databases (hash lattices) and replacing $\mathcal{T}$ by a network of insecure peers. The work of Schneier and Kelsey is referred to by numerous other researchers. The complete protocol has been implemented in a hardware token by Chong *et al.* [35] in 2002. Schneier and Kelsey have patented their work in 1997 [105].

While not stated explicitly in their work, Schneier and Kelsey actually designed a very simple version of a searchable encrypted log. By adding the permission mask

W_i (which they actually do not specify in detail), they add meta-data about the intended audience of the log entry. This leads to a more advanced set of secure logging mechanisms, in which searchability, privacy and auditability play a larger role, and in which more advanced cryptographic primitives and other tools are used.

3.2.2 Searchability and Privacy

In the setting of Schneier and Kelsey, a semi-trusted verifier $\mathcal{V}$ has access to a certain part of the log. The access control mechanism is enforced by the trusted server $\mathcal{T}$, who holds the initial authentication key and can therefore decrypt the entire log. Moreover, it is the *untrusted* log server $\mathcal{U}$ that decides which verifiers should get access to the log entries. In 2004, Waters *et al.* [119] proposed a new method for a searchable encrypted log. They also mention the individual's privacy as a sensitivity issue in log files, and a reason to encrypt the log entries. Their alternative to decrypting the entire log for searching, is adding protected keywords to the log entries, and introducing a (trusted) audit escrow agent to construct keyword search capabilities for (semi-trusted) investigators. Waters *et al.* provide a symmetric and an asymmetric version of their scheme. In the symmetric version, the log entries R_i in their system are structured as follows:

$$R_i = (\texttt{Enc}_{K_i}(D_i), \texttt{Hash}(R_{i-1}), (c_{w_a}, c_{w_b}, c_{w_c}, \ldots)),$$

where the message to be logged D_i is encrypted under the randomly generated secret key K_i, $H(R_{i-1})$ is part of the usual hash chain through the previously logged events, and $c_{w_a}, \ldots$ contain information representing the keywords and the encryption key K_i, generated using a master secret S, shared between the log server and the trusted audit escrow agent. Using the information in $c_{w_a}, \ldots$ and the assistance of the escrow agent, a verifier will be able to decrypt only those entries that contain the keywords that he or she presented to the escrow agent. A major problem with this approach is the fact that the escrow server has to share symmetric keys with all of his or her depending log servers. This problem is solved in the asymmetric scheme of Waters *et al.* [119], in which they use Identity-Based Encryption[2] (IBE). In their IBE setting, only the escrow agent holds the master secret of the IBE, and keywords are used as

[2] In IBE, any sequence of bits can be used as a public key. The corresponding private key can be generated using a master secret. Typically, an identifier (e.g, an email address) is used as the public key, hence the name *Identity*-Based Encryption [28].

public keys to encrypt a random symmetric key K that protects the actual log entry:

$$R_i = (\mathrm{Enc}_K(D), (c_{w_a}, c_{w_b}, c_{w_c}, \ldots)), \text{where } c_{w_\alpha} = \mathrm{IBE}_{w_\alpha}(\mathbf{flag}|K).$$

Given access to the log database, a verifier $\mathcal{V}$ contacts the audit escrow agent to get a decryption key d_w for the keyword w. For each log entry, $\mathcal{V}$ tries to IBE-decrypt each c_α. When this decrypts to **flag** followed by a random string, the verifier can retrieve K and decrypt the log entry. A major disadvantage of the asymmetric scheme is the overhead of the IBE. Waters *et al.* [119] propose some optimisations to counter this, in the area of the IBE itself (reusing intermediate results in the computation), and by grouping blocks of log entries with overlapping sets of keywords. Another major disadvantage of both the symmetric and the asymmetric scheme is that they assume a complete retrieval of the log database.

A more conventional approach is taken by Bergadano *et al.* [20] in their work from 2007, in which they describe a logging service with a set of auditors. They propose a symmetric as well as an asymmetric version of their system. In their schemes, each entry in the log file is encrypted under a random symmetric key, which is then encrypted with the public or secret key of the intended audience. The log entry is also included in a hash chain, and each hash chain value is signed by the log server. Each log entry can also be time-stamped. Finally, they discuss the possibility of group auditing through secret sharing schemes.

In 2005, Accorsi [1] proposed a variation on the scheme of Schneier and Kelsey, also making the distinction between *device* and *collector*. In Accorsi's scheme the device is generating the log entries, while the collector is recording/storing them. So-called *relays* are used as intermediates. In a later paper [6], Accorsi and Hohl discuss how parts of the computational tasks at the devices side can be delegated to a relay, by depending on a Trusted Platform Module (TPM) [65]. This way, resource-constraint devices can also use the secure log service. The privacy aspect of a secure logging system, again similar to the scheme of Schneier and Kelsey, is discussed by Accorsi in [2]. In this paper, Accorsi defines so-called *inner privacy* and *outer privacy*, which depend on the underlying threat model. Inner privacy is focused on the analysis of and tampering with private log data and is therefore protected by secure logging. In outer privacy, one tries to protect against observations of actions, e.g., the release of personal data. Typical technologies to protect outer privacy diminish the individual's observability (e.g., Tor [45]) and identity management systems that reduce and control the release of data (e.g., idemix [33]).

3.2.3 Maturing Secure Logs

As secure logging systems evolved, the concept of using the log itself to add meta-data about its structure was conceived. Moreover, public verifiability and the distribution of log services across several servers became topics of interest. In [64], Holt proposed to use the log itself to add structural information. He introduces Logcrypt, which further builds on the scheme by Schneier and Kelsey, adding public verifiability, aggregation of log entries within a log server, and aggregation of log databases across several log servers. The public key version of Logcrypt replaces hash chains by signatures. For each log entry, a new private key is used for which the corresponding public key is stored in a previous logged entry. This also establishes a chain through the logged entries:

$$R_i = (D_i, \texttt{Sign}_{\texttt{sk}_i}(D_i)),$$

with D_i the message to be logged, and $\texttt{sk}_i$ a private key. Encryption of the logged content D_i is only discussed in the symmetric version of Logcrypt. For each log entry, a new signature key-pair is used. Sets of key-pairs are generated periodically, and their public (verification) keys are recorded in a log entry called a meta-entry, which is signed by the last private key of the previous key-pair set. Holt also provides an alternative to generating huge amounts of key-pair sets, by proposing the use of an identity-based *signature* (IBS) scheme[3]. In this case, for each set of n key-pairs, new domain parameters are generated and logged in the meta-entry. The public (verification) keys are set to the numbers 1 to $n-1$, while the corresponding private keys are extracted with the master secret of the IBS scheme.

In [110] and [111], Stathoplousos *et al.* describe a more extensive security model, and list some precise requirements to which a secure log must comply. They also introduce a so-called *regulatory authority*, which receives periodic log file signatures from log servers, and provide an informal but clear security analysis of their design. The system of Stathopoulos *et al.* is also based on the Schneier and Kelsey scheme, with the addition of periodic digital signatures over the log database, computed by the log server, and verified and stored by the regulatory authority.

In more recent work [75], Ma and Tsudik use their earlier work on forward-secure sequential aggregate (FssAgg) signatures [73, 74] to solve two attacks on the Schneier and Kelsey scheme, which they describe in detail. The two attacks are as follows:

[3]Similar to ID-based encryption, any string can be a public key in ID-based signatures.

Truncation attack This attack refers to the fact that a set of log entries, residing on an untrusted log server $\mathscr{U}$ can be truncated without being detected, as long as no synchronisation with the trusted server $\mathscr{T}$ takes place.

Delayed detection This attack is possible because of the specifics of the verification protocol. In this protocol, a verifier $\mathscr{V}$ will pass only the last MAC (Z_l in the scheme) to the trusted server $\mathscr{T}$, to verify the logs integrity. However, because an attacker is supposed to have access to the authentication key $A_t, t < l$ when he compromises $\mathscr{U}$, he or she can safely generate a valid MAC. Independent of this, he or she can change the logged values $\texttt{Enc}_{K_i}(D_i), l_0 < i < l$ (into junk, as they are encrypted), with their corresponding hash chain values Y_i, where l_0 is the index of the last log entry L_{l_0} that was submitted to $\mathscr{T}$. Of course, once a synchronisation $\mathscr{U} - \mathscr{T}$ takes place, the attack is detected.

To tackle these problems, Ma and Tsudik propose a private- and public-verifiable scheme:

The private-verifiable scheme The scheme is a simplification of the scheme of Schneier and Kelsey, but with two hash chains; one for the verifier $\mathscr{V}$ and one for the trusted server $\mathscr{T}$, each with its own initial authentication key. For each chain, only the last value is kept, which is why a truncation or delayed detection attack is simply impossible. To ensure individual log verifiability, one MAC per log entry is added for the verifiers.

The public-verifiable scheme The scheme is based on one out of three public-key FssAgg schemes. In this scheme, a certification authority certifies a set of public keys used for the verification of the logged entries. Then, for each individual log entry, a signature is computed. Additional to this signature, periodic "umbrella" signatures are added to so-called anchor points, allowing verifiers to verify individual log entries or the entire log if they prefer to do so. The public-verifiable scheme of Ma and Tsudik is related to the solution by Holt: one of the FssAgg (BLS) schemes is based on bilinear maps, showing resemblance to identity-based encryption primitives.

In 2009, Yavuz and Ping [123] proposed the BAF (Blind-Aggregate-Forward) logging scheme, describing a new way of making a logging scheme publicly verifiable. To enable this, a trusted third-party generates an initial private signature key and a chain of public keys related to every evolution of the initial private key. The private key is securely transmitted to the log generator, while the public keys are

distributed to log verifiers. One of the novelties in the scheme is that the operations for generating log entries require low computational power; they are limited to a couple of additions and multiplications in a finite field. Due to the fact that they use such custom-built primitives they provide proofs for the soundness of their scheme. In their published solution, Yavuz and Ping only provide an all-or-nothing verification; i.e., a signature covering an entire round (set) of logged entries, similar as Ma and Tsudik in their solution. In later work [124], Yavuz, Ping and Reiter tackle this problem by proposing the FI-BAF (Fast-Immutable BAF) scheme, introducing individual signatures for each log entry. Because of the fact that their solution does not use standard signature primitives, the solution is a factor 100 times faster than conventional secure log services such as Holt's Logcrypt [64] and Ma and Tsudik's FssAgg-based schemes [75].

Related to his earlier work, Accorsi published two papers [3, 4] in 2009 in which he reviews some of the secure log systems described above, and classifies them with respect to their usability for digital evidence. He defines requirements for the transmission phase (log messages in transit) and the storage phase (log entries in a log server). The reviewed log systems are evaluated to these requirements and a set of open research problems are listed. In a subsequent paper [5] in 2011, Accorsi proposes BBox, a secure log system that covers both the transmission phase and the storage phase. The solution bears some resemblance to several earlier secure log schemes:

- The integrity of the log is protected by a hash chain.
- All log entries are encrypted with a freshly generated symmetric key.
- Hashed keywords are added to the log entries, to enable searching in the log.
- Intermediate hash chain values are signed using a conventional asymmetric signature primitive.

BBox allows for individual log entry verification and verification of the complete log file. To ensure that the log server is not tampered with, it uses remote attestation provided by secure hardware (a TPM). Accorsi also implemented BBox, including the TPM part. From his results, it shows that a moderate load of adding 1000 log entries/minute is easily feasible in a non-optimised (Java) implementation, on a standard PC.

3.3 Our Model

This section first provides an overview of the setting and our adversary model within the setting. Next, we present high-level goals that are then formalised.

3.3.1 Setting and Adversary Model

Figure 3.3 depicts a very simple distributed process. As processing is taking place for Alice's data, the controller Bob continuously sends data describing its data processing to its server that turns the data into *log entries* for Alice. Each log entry at a server concerns one user. At some point in time, Bob wishes to *fork* the process to a *downstream* controller Dave. When a process forks, the user is not involved, yet transparency logging of data processing should continue, potentially to a different server. We allow the structure of the process to be *dynamic*, meaning that the structure of the process is not necessarily predetermined. When Alice disclosed her data to Bob, neither Alice nor Bob needed to be aware of the fact that Bob would later share Alice's data with Dave. To realise this, we store metadata in the user's log when a fork takes place that enables the user to follow each fork in the process. In this case, when Bob shared the data with Dave, he logged some metadata for Alice. The exact structure of this metadata is later presented in Section 3.4.5. All of these log entries with metadata, spread across different servers, form a *log trail* between servers that only the user can reconstruct.

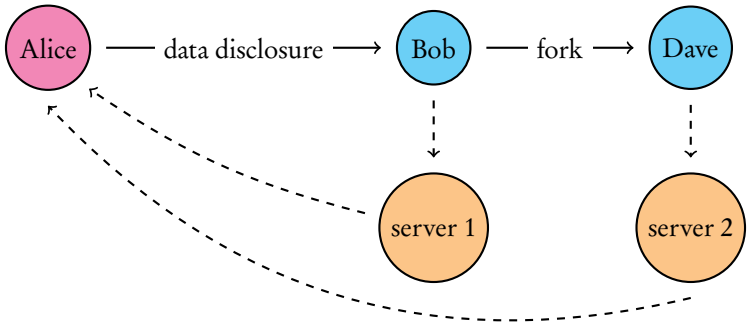


Figure 3.3 Setting for user *Alice*, controller *Bob*, and downstream controller *Dave*.

We consider the properties provided by the scheme for log entries that are *committed prior to compromise* of one or more servers and/or controllers. The users are fully trusted. In other words, we initially trust all entities in the scheme up to a point in

time t when an adversary compromises one or more servers and/or controllers. This is the standard adversary model within the secure logging area [5, 64, 75, 104, 123]. Servers keep some *state* information, apart from the storage needed for the log entries themselves. When a server is compromised, the adversary has full access to its internal state and storage. Our goal is to achieve strong cryptographic properties for all created log entries, and the metadata kept to create them (state), before t.

We argue that assuming initial trust in a controller is reasonable, since without (some degree of) initial trust, the user will not disclose data to the controller in the first place. Assuming that the controller is trustworthy, its automated data processing should be setup to automatically log data processing through our tool. When the controller becomes compromised, either by an external party or by an insider such as an employee, all descriptions of data processing that have been made up to that point should fulfill the goals outlined in the following sections. Little guarantee can be given about future processing once compromised. For servers, the required initial trust can be minimised by introducing trusted hardware at the server, e.g., as described by Vliegen *et al.* [118] (later summarised in Chapter 4). We also stress that auditability is an important property of a logging scheme, since the relationship between the server and the controller is a fragile one: the controller trusts the server to log as instructed, for the correct user, while the server trusts the controller to encrypt for the correct user. However, both trusted hardware and auditability are outside the scope of this chapter.

For communication, we assume secure channels between all entities. Furthermore, between users and servers we assume an anonymous channel.

3.3.2 Goals

We present four high-level goals for our scheme and briefly motivate why they are important in our setting. These goals are refinements of some of the requirements presented in Chapter 2, as we later discuss in Section 3.7. In Section 3.3.4, these are formally defined and in Section 3.5 we prove that the scheme presented in Section 3.4 realises these.

G1 The adversary should not be able to make undetectable modifications to logged data (committed prior to compromise), i.e., the *integrity* of the data should be ensured once stored.

G2 Only the user should be able to read the data logged for him or her, i.e., the data should be *confidential* (secrecy).

G3 From the log and the state, it should not be possible for an adversary to determine if two log entries (committed prior to compromise) are related to the same user or not, i.e., *user log entries* should be *unlinkable*.

G4 *User identifiers* (setup prior to compromise) across multiple controllers or servers should be *unlinkable*.

G1 and G2 are the basic notions of forward integrity and secrecy from the secure logging area, ensuring that logged data cannot be modified or read by an adversary. G3 and G4 emerge primarily as a consequence of our setting, where having multiple *recipients* of logged data leads to the need to protect the *privacy* of the recipient users. G3 protects the privacy of the user by preventing log entries from being linked to a particular user. If log entries could be linked together, patterns on log trails can reveal sensitive information. For example, the number of log entries in a trail can serve as a unique fingerprint of an event, generating a link to a controller or a user. G4 removes the links between each step (fork) in the process.

3.3.3 Notation

We denote the security parameter as $k \in \mathbb{N}$. A function $f : \mathbb{N} \to \mathbb{R}$ is 'polynomial' in k if $f(k) = O(k^n)$, with $n \in \mathbb{N}$. A function is 'negligible' if, for every $c \in \mathbb{N}$ there exists an integer k_c such that $f(k) \leq k^{-c}$ for all $k > k_c$. We denote a negligible function by $\epsilon(k)$. $response \leftarrow \texttt{Oracle}(args)$ denotes an algorithm or oracle with input $args$ that outputs $response$. An adversary $\mathcal{A}$ that outputs $response$ after making an arbitrary number of queries (in arbitrary order) to the listed oracles is denoted as $response \leftarrow \mathcal{A}^{\texttt{oracle}_1,\texttt{oracle}_2,\texttt{oracle}_3}()$. $\mathbf{Exp}^{property}_{\mathcal{A}}(k)$ is the experiment (in the security parameter k) that the challenger sets up for an adversary $\mathcal{A}$ against property $property$. $\mathbf{Adv}^{property}_{\mathcal{A}}(k)$ is the advantage of the adversary in breaking property $property$ (for a given security parameter). The advantage is between 0 and 1 for computational problems, between 0 and 1/2 for distinguishable problems[4].

3.3.4 Model and Definitions

To setup a system, we define two general algorithms:

- E ← `GenerateIdentifier`(): To generate a random, new and suitable identifier for an entity E within the logging scheme.

[4]The probability of guessing is already 1/2.

- `SetupEntity(S,E)`: To setup the state at the server S for entity E.

Without loss of generality, we assume a single controller C and a single server S. The controller has been initialised at the server by executing `SetupEntity(S,C)`. Users are added dynamically by the adversary. The server maintains a state for the controller and every user. The entire log is accessible to the adversary at all times. Let $\mathscr{A}$ denote the adversary that can adaptively control the system through a set of oracles:

- $\{U_i, l\} \leftarrow$ `CreateUser(`λ`)`: This oracle calls E $\leftarrow$ `GenerateIdentifier()` and runs `SetupEntity(S,`U_i`)` to setup the state for a new user at the server. Creates an initial log entry l for U_i by calling `CreateEntry(`U_i, λ`)` with the given data λ. The generated user identifier and log entry are returned.
- $l \leftarrow$ `CreateEntry(U,`λ`)`: This oracle creates a log entry l at server S for controller C and user U with the data λ. The log entry is returned.
- {`State`, `#entries`} $\leftarrow$ `CorruptServer()`: Corrupt the server S, which returns the entire state of the server and the number of created log entries before calling this oracle.

Note that, for properties with prefix forward (introduced shortly), the adversary cannot invoke any further queries after it makes a call to the `CorruptServer` oracle.

Forward Integrity

We adopt the definitions of Bellare and Yee [18] for forward integrity (FI) and deletion-detection FI secure logging schemes. FI captures that at the time of comprising the server, no log entries before that time can be forged without being detected. To ensure that no modifications to the log prior to compromise of the server can be made, truncation attacks (see Section 3.2) also need to be taken into account. For this reason deletion-detection FI is required.

First we define what constitutes a valid log entry: `valid`(l,i) returns whether or not the full log trail for every user verifies when l_i (the log entry created at the i-th call of `CreateEntry`) is replaced by l. Now, we are ready to present the experiment that the challenger sets up for $\mathscr{A}$ attacking the forward integrity (FI):
$\mathbf{Exp}^{FI}_{\mathscr{A}}(k)$:

1. $l \leftarrow \mathscr{A}^{\texttt{CreateUser},\texttt{CreateEntry},\texttt{CorruptServer}}()$
2. Return $l \neq l_i \wedge$ `valid`$(l,i) \wedge i \leq$ `#entries`.

The advantage of the adversary is defined as

$$\mathbf{Adv}^{FI}_{\mathscr{A}}(k) = Pr\left[\mathbf{Exp}^{FI}_{\mathscr{A}}(k) = 1\right].$$

Definition 1. *A logging system provides computational forward integrity, if and only if for all polynomial time adversaries $\mathscr{A}$, it holds that $\mathbf{Adv}^{FI}_{\mathscr{A}}(k) \leq \epsilon(k)$.*

Definition 2. *A logging system provides computational deletion-detection forward integrity, if and only if it is FI secure and the log verifier can determine, given the output of the log system and the time of compromise, whether any prior log entries have been deleted.*

Secrecy

The adversary $\mathscr{A}$ aiming to compromise the secrecy of the data contained within the log entries (SE), has to guess the bit b, for the modified `CreateEntry` oracle:

- $l \leftarrow$ `CreateEntry(U,`λ_0, λ_1`)`$_b$: This oracle creates a log entry l at server S for controller C and user U with the data λ_b. The log entry is returned.

$\mathbf{Exp}^{SE}_{\mathscr{A}}(k)$:

1. $b \in_R \{0,1\}$
2. $g \leftarrow \mathscr{A}^{\texttt{CreateUser,CreateEntry,CorruptServer}}()$
3. Return $g \stackrel{?}{=} b$.

The advantage of the adversary is defined as

$$\mathbf{Adv}^{SE}_{\mathscr{A}}(k) = \frac{1}{2} \cdot \left| Pr\left[\mathbf{Exp}^{SE}_{\mathscr{A}}(k) = 1 | b = 0\right] + Pr\left[\mathbf{Exp}^{SE}_{\mathscr{A}}(k) = 1 | b = 1\right] - 1 \right|.$$

Definition 3. *A logging system provides computational secrecy of the data contained within the log entries, if and only if for all polynomial time adversaries $\mathscr{A}$, it holds that $\mathbf{Adv}^{SE}_{\mathscr{A}}(k) \leq \epsilon(k)$.*

Forward Unlinkability of User Log Entries

The goal of the adversary $\mathscr{A}$ attacking the forward unlinkability of user log entries (FU) is to guess the bit b, for the modified `CreateEntry` oracle:

- $vuser \leftarrow$ `DrawUser`(U_i, U_j): This oracle generates a virtual user reference, as a monotonic counter, $vuser$ and stores $(vuser, U_i, U_j)$ in a table $\mathscr{D}$. If U_i is already referenced as the left-side user in $\mathscr{D}$ or U_j as the right-side user, then this oracle returns $\perp$ and adds no entry to $\mathscr{D}$. Otherwise, it returns $vuser$.

- `Free`$(vuser)$: this oracle removes the triple $(vuser, U_i, U_j)$ from table $\mathscr{D}$.

- $l \leftarrow$ `CreateEntry`$(vuser, \lambda)_b$: From the table $\mathscr{D}$, this oracle retrieves the corresponding (U_i, U_j). Depending on the value of b, $vuser$ either refers to U_i or U_j. A log entry l is created at server S for controller C and user U_i (if $b = 0$) or U_j (if $b = 1$) with the data λ. The log entry is returned.

Note that logging a message for a specific user U_i is still possible. This is required for the `CreateUser` oracle, that needs to log a message for the created user. By calling $vuser \leftarrow$ `DrawUser`(U_i, U_i), $vuser$ refers to U_i, regardless of the bit b.
$\mathbf{Exp}^{FU}_{\mathscr{A}}(k)$:

1. $b \in_R \{0,1\}$
2. $g \leftarrow \mathscr{A}^{\texttt{CreateUser},\texttt{DrawUser},\texttt{Free},\texttt{CreateEntry},\texttt{CorruptServer}}()$
3. Return $g \stackrel{?}{=} b$.

The advantage of the adversary is defined as

$$\mathbf{Adv}^{FU}_{\mathscr{A}}(k) = \frac{1}{2} \cdot \left| Pr\left[\mathbf{Exp}^{FU}_{\mathscr{A}}(k) = 1 | b = 0\right] + Pr\left[\mathbf{Exp}^{FU}_{\mathscr{A}}(k) = 1 | b = 1\right] - 1 \right|.$$

Definition 4. *A logging system provides computational forward unlinkability of user log entries, if and only if for all polynomial time adversaries $\mathscr{A}$, it holds that $\mathbf{Adv}^{FU}_{\mathscr{A}}(k) \leq \epsilon(k)$.*

Unlinkability of User Identifiers

This property implies multiple controllers, hence the oracles `CreateUser` and `CreateEntry` need to be made controller dependent. Without loss of generality, we can still assume that there is a single server and that all controllers have been initialised at this server. An attacker against Unlinkability of User Identifiers (UU) has to guess the bit b, used in the `Fork` oracle:

- $\{U', l\} \leftarrow \texttt{Fork}(U, C_0, C_1)_b$: For a user U setup with controller C_0, this oracle constructs U' either by randomising $U (b = 0)$ or by generating a new identifier by calling `GenerateIdentifier()` $(b = 1)$. Next, U' is setup for C_1 at server S. The resulting data, together with the data used to randomise the identifier $(b = 0)$ or random data of equal length $(b = 1)$, is logged for user U and controller C_0. The new user and log entry is returned.

$\mathbf{Exp}^{UU}_{\mathscr{A}}(k)$:

1. $b \in_R \{0,1\}$
2. $g \leftarrow \mathscr{A}^{\texttt{CreateUser},\texttt{CreateEntry},\texttt{Fork},\texttt{CorruptServer}}()$
3. Return $g \stackrel{?}{=} b$.

The advantage of the adversary is defined as

$$\mathbf{Adv}^{UU}_{\mathscr{A}}(k) = \frac{1}{2} \cdot \left| Pr\left[\mathbf{Exp}^{UU}_{\mathscr{A}}(k) = 1 | b = 0\right] + Pr\left[\mathbf{Exp}^{UU}_{\mathscr{A}}(k) = 1 | b = 1\right] - 1 \right| .$$

Definition 5. *A logging system provides computational unlinkability of user identifiers, if and only if for all polynomial time adversaries $\mathscr{A}$, it holds that $\mathbf{Adv}^{UU}_{\mathscr{A}}(k) \leq \epsilon(k)$.*

3.4 Our Scheme

This section describes our proposed scheme. First, we explain the idea behind the log entry structure and the state kept at the server. Then, we introduce the required cryptographic building blocks. Finally, we describe the components of our proposal in detail.

3.4.1 Log Entry Structure and State

A log entry is created by a server for a user U when the controller C sends some data to log. A log entry consists of five fields:

Data The data field contains the actual data to be logged in an encrypted form, such that only the user can derive the plaintext.

$\mathbf{IC_U}$ The *index chain* field for the user serves as an identifier for the log entry for the user. The values of this field create a chain that links all log entries for the user together. Only the user can reconstruct this chain.

DC_U The *data chain* field for the user allows the user to verify the validity of this log entry. All entries that were created for this user are chained together, leading to *cumulative verification*, i.e., verifying the integrity of one log entry verifies all the previous log entries too.

IC_C The index chain field for the controller.

DC_C The data chain for the controller.

The controller supplies the value that goes into the data field, while the index chain fields are derived from the state kept by the server. The data chain fields are derived from the state kept by the server, the data field and index fields from the log entry. A server keeps the following values in its state *for each* user and controller:

AK The current authentication key, used as a key when generating the DC field of the next log entry, and when updating state after log entry creation.

IC An intermediate index chain value, used to generate the next IC field of a log entry, derived from the IC field of the previous log entry for this entity.

DC An intermediate data chain value, used to generate the next DC field of a log entry, derived from the DC field of the previous log entry for this entity.

The state is *updated* each time a log entry is created, making it hard to recover the previous values stored in state. This is the mechanism at the core of the *prior to compromise* adversary model in our setting. Due to the fact that the state kept by servers is continuously updated as log entries are created, the adversary is unable to reconstruct prior states needed for manipulating log entries created prior to compromise. Figure 3.4 illustrates the interplay between log entries, the server's state, and our adversary model.

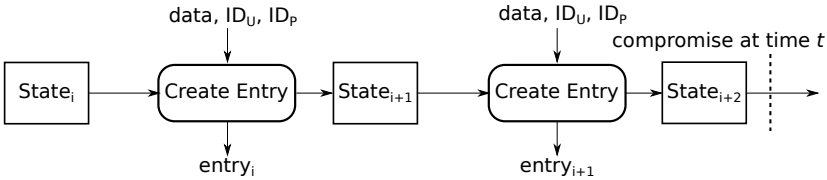


Figure 3.4 The interplay between log entries, the server's state, and our adversary model.

3.4.2 Cryptographic Building Blocks

To provide forward integrity and forward unlinkability of user identifiers, we make use of a cryptographic hash function $\texttt{Hash}(\cdot)$ and a cryptographic message authentication code generation algorithm $\texttt{MAC}_K(\cdot)$ under a secret key K. Each time a log entry is created the secret key K used for the MAC algorithm is evolved using the hash function. For the MAC algorithm, we make use of the general HMAC construction.

In our scheme, the identifiers of entities are public keys that can be used to encrypt messages for this entity. An encryption scheme $\prod$ consists of three algorithms: $\texttt{GenerateKey}()$, $\texttt{Enc}_{\texttt{pk}}(p)$, and $\texttt{Dec}_{\texttt{sk}}(c)$. $\texttt{GenerateKey}()$ generates a new random key-pair $\{\texttt{sk},\texttt{pk}\}$ consisting of public key $\texttt{pk}$ and private key $\texttt{sk}$. $\texttt{Enc}_{\texttt{pk}}(p)$ outputs the encryption of a plaintext p with public key $\texttt{pk}$. $\texttt{Dec}_{\texttt{sk}}(c)$ for a key-pair $\{\texttt{sk},\texttt{pk}\}$, outputs the plaintext p by decrypting the ciphertext $c \leftarrow \texttt{Enc}_{\texttt{pk}}(p)$. For our logging scheme to provide secrecy, $\prod$ should provide indistinguishability under chosen-plaintext attack (IND-CPA) security [17]. Furthermore, for our logging scheme to achieve forward unlinkability of user identifiers, $\prod$ should also be key private under chosen plaintext attack (IK-CPA) [16]. For efficiency, a hybrid cipher with asymmetric key encapsulation and symmetric data encapsulation mechanism (KEM-DEM) [63] will be used. We selected the elliptic curve integrated encryption scheme (ECIES) [109], as defined in ISO/IEC 18033-2. User identifiers are generated as follows: $\{\texttt{sk},\texttt{pk}\} \leftarrow \texttt{GenerateKey}()$, where $\texttt{sk} = x \in_R \mathbb{Z}_n$ and $\texttt{pk} = X = xG$, for a generator G of the curve.

The controller and server have separate key-pairs used for signing purposes, in order to establish the origin of messages. A signature scheme consists of three algorithms: $\texttt{GenerateKey}()$, $\texttt{Sign}_{\texttt{sk}}(m)$, and $\texttt{Verify}_{\texttt{pk}}(\sigma, m)$. $\texttt{GenerateKey}()$ generates a random, new key-pair $\{\texttt{sk},\texttt{pk}\}$. $\texttt{Sign}_{\texttt{sk}}(m)$ outputs a signature σ on message m using private key $\texttt{sk}$. $\texttt{Verify}_{\texttt{pk}}(\sigma, m)$ verifies that σ is a valid signature on m using the public key $\texttt{pk}$, and returns true if the signature is valid, otherwise false. The signature scheme should be non-repudiable and selectively unforgeable under known message attack [58]. We selected ECDSA [85].

3.4.3 Setup and Stopping

Before a controller can start logging data for users, it needs to be set up at its server. First the controller generates a key-pair and uses the public key as its identifier at its server. The server runs Algorithm 1 with the controller's public key as input and returns the output to the controller. This algorithm initialises the server's state for

the provided identifier and outputs the initial *authentication key* (AK_0) encrypted under this identifier. To guarantee the origin of the initial authentication key, it is signed by the server prior to encryption. The controller decrypts the output of the server, verifies the signature and stores AK_0.

Algorithm 1 Setup of an entity at a server.

Require: An entity identifier ID_E.
Ensure: The encrypted initial values in `State`.
1: $AK_0 \leftarrow \mathtt{Rand}(|\mathtt{Hash}(\cdot)|)$
2: $\mathtt{State}(\mathrm{ID}_\mathrm{E}, IC) \leftarrow \mathtt{MAC}_{AK_0}(\mathrm{ID}_\mathrm{E})$
3: $\mathtt{State}(\mathrm{ID}_\mathrm{E}, DC) \leftarrow$ null
4: $\mathtt{State}(\mathrm{ID}_\mathrm{E}, AK) \leftarrow \mathtt{Hash}(AK_0)$
5: **return** $\mathtt{Enc}_{\mathrm{ID}_\mathrm{E}}\big(AK_0, \mathtt{Sign}_{\mathtt{sk}_\mathrm{S}}(AK_0, \mathrm{ID}_\mathrm{E})\big)$

To set up users, the user first generates a fresh user identifier as follows:

$$\mathrm{ID}_\mathrm{U} \leftarrow \mathtt{pk}, \text{where } \{\mathtt{sk}, \mathtt{pk}\} \leftarrow \mathtt{GenerateKey()} \tag{3.1}$$

Then, the user runs Protocol 1 to set up transparency logging at the controller. At the end of the protocol, the user will be in possession of the initial authentication key, and be assured that it was generated by the server. This initial authentication key will be used by the user to download and verify all log entries related to him from the server, as described later in Section 3.4.6. The controller keeps track of the association between the data that the user disclosed and the user identifier ID_U such that the controller can later log data processing for the correct user.

Protocol 1 Setup between user U and controller C with server S.

Require: C is set up at S, user identifier ID_U.
Ensure: User knowns AK_0 for ID_U.
1: $\mathrm{U} \longrightarrow \mathrm{C} \longrightarrow \mathrm{S} : \mathrm{ID}_\mathrm{U}$
2: $\mathrm{S} : \alpha \leftarrow \mathtt{Enc}_{\mathrm{ID}_\mathrm{U}}\big(AK_0, \mathtt{Sign}_{\mathtt{sk}_\mathrm{S}}(AK_0, \mathrm{ID}_\mathrm{U})\big) \leftarrow$ Algorithm 1 with input ID_U
3: $\mathrm{S} \longrightarrow \mathrm{C} : \alpha$
4: $\mathrm{C} \longrightarrow \mathrm{U} : \alpha, \mathtt{Sign}_{\mathtt{sk}_\mathrm{C}}(\alpha)$

When a controller has finished processing on a user's data, the controller constructs one final log entry for the user with the marker M_S that signals to the user

that the controller has finished its processing. We describe how to create a log entry in Section 3.4.4. Next, the controller instructs the server to delete the state kept for that user from the server.

3.4.4 Generating Log Entries

When a controller performs processing on a user's disclosed data, it logs a description of the processing to the log trail of the user located at the server used by the controller as described in Protocol 2. The controller first signs the data to log (to prove the origin to the user) and then encrypts the data and signature under the public key of the user. Next, the controller sends the resulting ciphertext to the server who creates a log entry.

Protocol 2 The generate log entry protocol for controller C with server S.

Require: The controller identifier $\mathrm{ID_C}$, user identifier $\mathrm{ID_U}$, and data λ to log.
Ensure: Log entry l created for $\mathrm{ID_U}$ and $\mathrm{ID_C}$ with data λ.

1: C: $d \leftarrow \mathtt{Enc}_{\mathrm{ID_U}}\big(\lambda, \mathtt{Sign}_{\mathtt{sk_C}}(\lambda)\big)$
2: $\mathrm{C} \longrightarrow \mathrm{S} : \mathrm{ID_C}, \mathrm{ID_U}, d$
3: $\mathrm{S} \longrightarrow \mathrm{C} : l \leftarrow$ Algorithm 2 with input $\mathrm{ID_C}, \mathrm{ID_U}$ and d

The server creates the log entry by running Algorithm 2. First, the log entry is created (steps 1–5) and written into storage (step 6). Next, the server's state is updated for both the involved user and controller (steps 7–12). In the process of updating the state, any old values are overwritten and no longer accessible afterwards. At the end (step 13), the generated log entry is returned.

3.4.5 Forking a Process

When a controller wishes to involve another controller in the processing of a user's data, the transparency logging needs to fork such that the data processing can be made transparent to the user at the additional controller. As part of forking, the public key used to identify the user needs to be *blinded*, to prevent linking the transparency logging on both controllers for the same user. Blinding a public key X is done by generating a random integer $r \neq 0$ in $\mathbb{Z}_n$, and used to generate a new public key X' as follows:

$$X' = X + rG = (x + r)G \tag{3.2}$$

Algorithm 2 The algorithm for creating a log entry.

Require: Identifiers for a controller $\mathrm{ID_C}$ and a user $\mathrm{ID_U}$, and the data d to log.
Ensure: Log entry created and stored.

$IC_{\mathrm{U}} \leftarrow \mathtt{Hash}\big(\mathtt{State}(ID_{\mathrm{U}}, IC)\big)$
$DC_{\mathrm{U}} \leftarrow \mathtt{MAC}_{\mathtt{State}(ID_{\mathrm{U}},AK)}\big(\mathtt{State}(ID_{\mathrm{U}}, DC), IC_{\mathrm{U}}, d\big)$
$IC_{\mathrm{C}} \leftarrow \mathtt{Hash}\big(\mathtt{State}(ID_{\mathrm{C}}, IC)\big)$
$DC_{\mathrm{C}} \leftarrow \mathtt{MAC}_{\mathtt{State}(ID_{\mathrm{C}},AK)}\big(\mathtt{State}(ID_{\mathrm{C}}, DC), IC_{\mathrm{C}}, DC_{\mathrm{U}}, IC_{\mathrm{U}}, d\big)$
$l_i \leftarrow \{IC_{\mathrm{U}}, DC_{\mathrm{U}}, IC_{\mathrm{C}}, DC_{\mathrm{C}}, d\}$
$\mathtt{Storage}(IC_{\mathrm{U}}) \leftarrow \mathtt{Storage}(IC_{\mathrm{C}}) \leftarrow l_i$
$\mathtt{State}(ID_{\mathrm{U}}, IC) \leftarrow \mathtt{MAC}_{\mathtt{State}(ID_{\mathrm{U}},AK)}(IC_{\mathrm{U}})$
$\mathtt{State}(ID_{\mathrm{U}}, DC) \leftarrow \mathtt{MAC}_{\mathtt{State}(ID_{\mathrm{U}},AK)}(DC_{\mathrm{U}})$
$\mathtt{State}(ID_{\mathrm{U}}, AK) \leftarrow \mathtt{Hash}\big(\mathtt{State}(ID_{\mathrm{U}}, AK)\big)$
$\mathtt{State}(ID_{\mathrm{C}}, IC) \leftarrow \mathtt{MAC}_{\mathtt{State}(ID_{\mathrm{C}},AK)}(IC_{\mathrm{C}})$
$\mathtt{State}(ID_{\mathrm{C}}, DC) \leftarrow \mathtt{MAC}_{\mathtt{State}(ID_{\mathrm{C}},AK)}(DC_{\mathrm{C}})$
$\mathtt{State}(ID_{\mathrm{C}}, AK) \leftarrow \mathtt{Hash}\big(\mathtt{State}(ID_{\mathrm{C}}, AK)\big)$
return l_i

The corresponding private key is $x + r$, which is easy to compute when knowing both the original secret key x and the blinding factor r.

Protocol 3 describes forking between two controllers, the initiating (that wishes to fork the process) C_a and the receiving C_b, that use servers S_a and S_b respectively, where S_a and S_b may or may not be the same server. First, C_a creates a blinded user identifier for the user using Equation (3.2) (step 1). Next, this new user is set up with controller C_b and server S_b, where C_a takes on the role of the user in Protocol 1 (step 2). Finally (step 3), C_a creates a special log entry at its server S_a for ID_{U} with M_f, a forking marker indicating that a fork has taken place, r, the information needed to generate the corresponding blinded private key, and β, the encrypted initial authentication key from the server used by the receiving controller. The user ID_{U} can later, with $\{M_f, r, \beta\}$, reconstruct the log trail for ID'_{U} used by controller C_b at server S_b.

3.4.6 Reconstruction

When the user disclosed data to the controller, the user initiated Protocol 1, generating an identifier $\mathrm{ID_U}$ in the process, and getting an initial authentication key AK_0 from

Protocol 3 The forking protocol between controllers C_a and C_b with servers S_a and S_b, respectively.

Require: The user identifier ID_U.
Ensure: Forking information written to the log.

1: $C_a : (ID'_U, r) \leftarrow$ Equation (3.2) with input ID_U
2: $C_a : \beta \leftarrow$ Protocol 1 with controller C_b and its server S_b for user identifier ID'_U
3: C_a : run Protocol 2 with S_a, for ID_U with data $\lambda = \{M_f, r, \beta\}$

the controller. To reconstruct the log trail, the user first downloads all log entries, stored at the server used by this controller, linked to his identifier ID_U. Next, the user validates the downloaded log trail.

Based on how a user is set up at the server (Algorithm 1), and how log entries are created (Algorithm 2), the following equations describe how the user can generate the index chain value of all the user's log entries, for the i:th log entry where $i \geq 1$, given AK_0 and ID_U:

$$IC_{U_1} = \texttt{Hash}\big(MAC_{AK_0}(ID_U)\big) \tag{3.3a}$$

$$AK_i = \texttt{Hash}(AK_{i-1}) \tag{3.3b}$$

$$IC_{U_i} = \texttt{Hash}\big(MAC_{AK_{i-1}}(IC_{U_{i-1}})\big) \tag{3.3c}$$

A server will provide any log entry it stores that matches a provided index chain value, so Equation (3.3) enables users to download all of their log entries at a server sequentially until the response from the server no longer contains any data. The straightforward way for a user to sequentially download log entries from a server may reveal (i) the order in which log entries were generated, and (ii) that log entries belong to the same user, i.e., one can link log entries together. Please note that user behaviour is not considered in our model since the negatively effects to the unlinkability properties of log entries also depend on the setting our system is deployed in. Users can however limit these effects by (i) randomising the order in which log entries are requested, at the cost of introducing some requests for log entries not yet created and (ii) waiting some time between each request for a log entry. Furthermore, users can cache already downloaded log entries and it is assumed that servers will be serving multiple controllers and users at the same time.

Protocol 4 describes the necessary steps to verify the authenticity of the log trail. First, the user verifies locally the authenticity of the downloaded log entries (steps 1–9). For each entry, the MAC in the DC_{U} field of the entry is compared to the correct computed value (step 4), and the data field of the entry is decrypted and then the signature of the controller on the logged data is verified (steps 6–8). Next, the user requests the IC_{U} value stored in the server's state for ID_{U} (steps 10–14). The server replies by sending the IC_{U} value (or, if it does not exist, indicating that logging has stopped, a random value of equal length) encrypted under the provided identifier. By sending this value encrypted, only the legitimate user will be able to learn the current IC_{U} value at the server. This value is checked against the last entry in the log trail (steps 15–18). This interaction with the server allows the user to detect deletion of log entries generated for his identifier (prior to compromise of the server).

Protocol 4 Verify the authenticity of a log trail at server S for user ID_{U}.

Require: The user's identifier ID_{U}, corresponding private key $\mathtt{sk}_{\mathrm{U}}$, initial authentication key AK_0, list of log entries log, and the controller's public key $\mathtt{pk}_{\mathrm{C}}$ for signature verification.

Ensure: True if the log trail in log is valid, false otherwise.

1: $index = 0$, $key \leftarrow AK_0$, $chain \leftarrow$ null
2: **while** $index < |log|$ **do**
3: $\quad e \leftarrow log[index], key \leftarrow \mathtt{Hash}(key)$
4: $\quad$ **if** $e.DC_{\mathrm{U}} \neq \mathtt{MAC}_{key}(chain, e.IC_{\mathrm{U}}, e.Data)$ **then**
5: $\quad\quad$ **return** false
6: $\quad (m, \sigma = \mathtt{Sign}_{\mathtt{sk}_{\mathrm{C}}}(m)) \leftarrow \mathtt{Dec}_{\mathtt{sk}_{\mathrm{U}}}(e.Data)$
7: $\quad$ **if** $\mathtt{Verify}_{\mathtt{pk}_{\mathrm{C}}}(\sigma, m) \neq$ true **then**
8: $\quad\quad$ **return** false
9: $\quad chain \leftarrow \mathtt{MAC}_{key}(e.DC_{\mathrm{U}}), index{+}{+}$
10: $\mathrm{U} \longrightarrow \mathrm{S} : \mathrm{ID}_{\mathrm{U}}$
11: **if** $\mathrm{S} : \mathtt{State}(\mathrm{ID}_{\mathrm{U}}, IC) \neq$ null **then**
12: $\quad \mathrm{S} \longrightarrow \mathrm{U} : r \leftarrow \mathtt{Enc}_{\mathrm{ID}_{\mathrm{U}}}\big(\mathtt{State}(\mathrm{ID}_{\mathrm{U}}, IC)\big)$
13: **else**
14: $\quad \mathrm{S} \longrightarrow \mathrm{U} : r \leftarrow \mathtt{Enc}_{\mathrm{ID}_{\mathrm{U}}}\big(\mathtt{Rand}(|\mathtt{MAC}(\cdot)|)\big)$
15: **if** m contains marker M_s **then**
16: $\quad$ **return** $\mathtt{Dec}_{\mathtt{sk}_{\mathrm{U}}}(r) \neq \mathtt{MAC}_{key}(e.IC_{\mathrm{U}})$
17: **else**
18: $\quad$ **return** $\mathtt{Dec}_{\mathtt{sk}_{\mathrm{U}}}(r) \stackrel{?}{=} \mathtt{MAC}_{key}(e.IC_{\mathrm{U}})$

After verification of the log trail, the user searches for any entries with the forking marker M_f. For each such entry, the user uses his private key $\mathtt{sk}$ and together with the blinding factor r from the log entry to construct the blinded private key $\mathtt{sk}'$ as described in Section 3.4.5. With the blinded private key, the user can decrypt the additional payload of the log entry (see Protocol 3) and recover the initial authentication key AK'_0 for the forked process. With this information, the user repeats the procedure outlined in this section to reconstruct the rest of the log trail.

Note that a controller could also retrieve all its created log entries from its server using a similar approach as described in this section, and verify the integrity of all data by computing the data chain for the controller.

3.5 Evaluation

In this section, we evaluate our scheme with respect to the security and privacy properties defined in Section 3.3.4.

3.5.1 Forward Integrity

Theorem 1. *In the random oracle model, the proposed logging scheme provides computational forward integrity, according to Definition 1.*

Proof. This follows directly from the proof of Theorem 3.2 by Bellare and Ye [19]. They showed that for a standard message authentication scheme and a forward secure pseudo-random generator, the general construction of a key-evolving message authentication scheme is forward secure. □

Theorem 2. *In the random oracle model, the proposed logging scheme provides computational deletion-detection forward integrity, according to Definition 2.*

Proof. First, the scheme provides computational forward integrity. Second, we will show that user can detect if log entries in his trail are deleted. We have to differentiate between two cases: (1) the logging for this user has ended and (2) the logging for this user is still ongoing. In the first case the user will be able to validate its entire log trail until the last entry containing a stopping marker. In absence of this marker in the last log entry, the user will assume to be in the second case and ask the server for the current $\mathtt{State}(\mathrm{ID}_\mathrm{U}, IC)$ value. This value is encrypted under the user's public key. Hence, this mechanism only allows the legitimate user to learn this value. Now we need to show that a corrupted server cannot produce a previous $\mathtt{State}(\mathrm{ID}_\mathrm{U}, IC)$

value, given the current state and all log entries. For an adversary to truncate the log entries for a certain user U given the last entry for this user to retain, $IC_{\mathrm{U}_{last}}$, and the first entry for this user after that, $IC_{\mathrm{U}_{truncated}}$, this adversary has to come up with $\texttt{State}_{last}(\mathrm{ID}_{\mathrm{U}}, IC)$ that follows $IC_{\mathrm{U}_{last}}$. The following relations exist:

- $\texttt{State}_{last}(\mathrm{ID}_{\mathrm{U}}, IC) = \texttt{MAC}_{\texttt{State}_{last}(ID_{\mathrm{U}}, AK)}(IC_{\mathrm{U}_{last}})$ and
- $IC_{\mathrm{U}_{truncated}} = \texttt{Hash}(\texttt{State}_{last}(\mathrm{ID}_{\mathrm{U}}, IC))$.

Given the one-way nature of the hash function and non-malleability of the MAC algorithm, the advantage of an adversary for creating a valid $\texttt{State}_{last}(\mathrm{ID}_{\mathrm{U}}, IC)$ is negligible. □

3.5.2 Secrecy

Theorem 3. *Under the Decisional Diffie-Hellman (DDH) assumption, the proposed logging scheme provides computational secrecy of the data contained within the log entries, according to Definition 3.*

Proof. The data field in the log entry, for a user U and data λ_b, is $d = \texttt{Enc}_{\mathrm{ID}_{\mathrm{U}}}\big(\lambda_b, \texttt{Sign}_{\texttt{sk}_{\mathrm{C}}}(\lambda_b)\big)$. For an adversary to distinguish between λ_0 and λ_1 being logged, another adversary with the same success probability can be constructed, breaking the IND-CPA security of the encryption scheme Enc. For the used ECIES encryption scheme, this probability is negligible under the DDH assumption. □

3.5.3 Forward Unlinkability of User Log Entries

Theorem 4. *In the random oracle model, under the DDH assumption, the proposed logging scheme provides computational forward unlinkability of user log entries, according to Definition 4.*

Proof. We use the game based proof technique proposed by Shoup [107]. The initial game corresponds to the proposed scheme. Only the differences with the previous game are explicitly mentioned.

Game 0:

- `CreateUser(`λ`)` creates a new user U_i and logs the first data λ for that user as follows:
  - $\mathrm{U}_i \leftarrow \texttt{pk} \leftarrow \texttt{GenerateKey()}$

- $AK_0 \leftarrow \texttt{Rand}(|\texttt{Hash}(\cdot)|)$
- $\texttt{State}(\mathrm{ID}_{\mathrm{U}_i}, IC) \leftarrow \texttt{MAC}_{AK_0}(\mathrm{ID}_{\mathrm{U}_i})$
- $\texttt{State}(\mathrm{ID}_{\mathrm{U}_i}, DC) \leftarrow \text{null}$
- $\texttt{State}(\mathrm{ID}_{\mathrm{U}_i}, AK) \leftarrow \texttt{Hash}(AK_0)$
- $\texttt{DrawUser}(\mathrm{U}_i, \mathrm{U}_i) \rightarrow vuser$
- $\texttt{CreateEntry}(vuser, \lambda) \rightarrow l$, return U_i, l.

- $\texttt{CreateEntry}(vuser, \lambda)$ creates a log entry l at server S for controller C and user U for given data λ as follows:
  - $(vuser, \mathrm{U}_i, \mathrm{U}_j) \leftarrow \mathscr{D}$, $\mathrm{U} = \mathrm{U}_i$ (if $b = 0$) or $\mathrm{U} = \mathrm{U}_j$ (if $b = 1$)
  - $d = \texttt{Enc}_{\texttt{pk}_\mathrm{U}}\big(\lambda, \texttt{Sign}_{\texttt{sk}_\mathrm{C}}(\lambda)\big)$
  - $IC_\mathrm{U} \leftarrow \texttt{Hash}\big(\texttt{State}(ID_\mathrm{U}, IC)\big)$
  - $DC_\mathrm{U} \leftarrow \texttt{MAC}_{\texttt{State}(ID_\mathrm{U}, AK)}\big(\texttt{State}(ID_\mathrm{U}, DC), IC_\mathrm{U}, d\big)$
  - $IC_\mathrm{C} \leftarrow \texttt{Hash}\big(\texttt{State}(ID_\mathrm{C}, IC)\big)$
  - $DC_\mathrm{C} \leftarrow \texttt{MAC}_{\texttt{State}(ID_\mathrm{C}, AK)}\big(\texttt{State}(ID_\mathrm{C}, DC), IC_\mathrm{C}, DC_\mathrm{U}, IC_\mathrm{U}, d\big)$
  - $\texttt{State}(ID_\mathrm{U}, IC) \leftarrow \texttt{MAC}_{\texttt{State}(ID_\mathrm{U}, AK)}(IC_\mathrm{U})$
  - $\texttt{State}(ID_\mathrm{U}, DC) \leftarrow \texttt{MAC}_{\texttt{State}(ID_\mathrm{U}, AK)}(DC_\mathrm{U})$
  - $\texttt{State}(ID_\mathrm{U}, AK) \leftarrow \texttt{Hash}\big(\texttt{State}(ID_\mathrm{U}, AK)\big)$
  - $\texttt{State}(ID_\mathrm{C}, IC) \leftarrow \texttt{MAC}_{\texttt{State}(ID_\mathrm{C}, AK)}(IC_\mathrm{C})$
  - $\texttt{State}(ID_\mathrm{C}, DC) \leftarrow \texttt{MAC}_{\texttt{State}(ID_\mathrm{C}, AK)}(DC_\mathrm{C})$
  - $\texttt{State}(ID_\mathrm{C}, AK) \leftarrow \texttt{Hash}\big(\texttt{State}(ID_\mathrm{C}, AK)\big)$
  - $l \leftarrow \{IC_\mathrm{U}, DC_\mathrm{U}, IC_\mathrm{C}, DC_\mathrm{C}, d\}$
  - return l.
- $\texttt{CorruptServer}()$ returns $\texttt{State}(\mathrm{ID}_\mathrm{C}, AK)$, $\texttt{State}(\mathrm{ID}_\mathrm{C}, IC)$, $\texttt{State}(\mathrm{ID}_\mathrm{C}, DC)$ and $\forall\, \mathrm{U}_i$: $\texttt{State}(\mathrm{ID}_{\mathrm{U}_i}, AK)$, $\texttt{State}(\mathrm{ID}_{\mathrm{U}_i}, IC)$, $\texttt{State}(\mathrm{ID}_{\mathrm{U}_i}, DC)$.
- Output of the experiment: $g \leftarrow \mathscr{A}$.

The event S_i is defined as $\mathscr{A}$ outputting a correct guess in game i, *i.e.* $g \stackrel{?}{=} b$.

$$Pr[S_0] = 1/2 \cdot \big(Pr\big[\mathbf{Exp}^0_{\mathscr{A}}(k) = 1\big] + Pr\big[\mathbf{Exp}^1_{\mathscr{A}}(k) = 1\big]\big)$$

Game 1: The encryption of the data and signature on it is done under a random encryption key.

- `CreateEntry(`$vuser$`,`λ`)`:
  - $d = \mathtt{Enc}_{\mathtt{pk}_R}\big(\lambda, \mathtt{Sign}_{\mathtt{sk}_C}(\lambda)\big)$ with $\mathtt{pk}_R \leftarrow$ `GenerateKey()`

The difference $|Pr[S_0] - Pr[S_1]|$ is the advantage of key privacy under chosen plaintext attack (IK-CPA) [16] of a distinguisher. For the used ECIES encryption scheme, this advantage is equal to the DDH advantage.

Game 2: The random oracle $\mathcal{O}^{RO}(\cdot)$ is used to replace the outputs of the hash function by random values. The $\mathtt{MAC}_K(m)$ algorithm in our proposed scheme is instantiated with $\mathrm{HMAC}(K,m) = \mathtt{Hash}\big((K \oplus opad)||\mathtt{Hash}\big((K \oplus ipad)||m\big)\big)$.

- `CreateUser(`λ`)`:
  - $\mathtt{State}(\mathrm{ID}_{\mathrm{U}_i}, IC) \leftarrow \mathcal{O}^{RO}\big((AK_0 \oplus opad)||\mathcal{O}^{RO}\big((AK_0 \oplus ipad)||\mathrm{ID}_{\mathrm{U}_i}\big)\big)$
  - $\mathtt{State}(\mathrm{ID}_{\mathrm{U}_i}, AK) \leftarrow \mathcal{O}^{RO}\big(\mathtt{State}(\mathrm{ID}_{\mathrm{U}_i}, AK)\big)$
- `CreateEntry(`$vuser$`,`λ`)`:
  - $IC_{\mathrm{U}} \leftarrow \mathcal{O}^{RO}\big(\mathtt{State}(ID_{\mathrm{U}}, IC)\big)$
  - $DC_{\mathrm{U}} \leftarrow \mathcal{O}^{RO}\big((\mathtt{State}(ID_{\mathrm{U}}, AK) \oplus opad)||\mathcal{O}^{RO}(\ldots)\big)$
  - $IC_{\mathrm{C}} \leftarrow \mathcal{O}^{RO}\big(\mathtt{State}(ID_{\mathrm{C}}, IC)\big)$
  - $DC_{\mathrm{C}} \leftarrow \mathcal{O}^{RO}\big((\mathtt{State}(ID_{\mathrm{C}}, AK) \oplus opad)||\mathcal{O}^{RO}(\ldots)\big)$
  - $\mathtt{State}(ID_{\mathrm{U}}, IC) \leftarrow \mathcal{O}^{RO}\big((\mathtt{State}(ID_{\mathrm{U}}, AK) \oplus opad)||\mathcal{O}^{RO}(\ldots)\big)$
  - $\mathtt{State}(ID_{\mathrm{U}}, DC) \leftarrow \mathcal{O}^{RO}\big((\mathtt{State}(ID_{\mathrm{U}}, AK) \oplus opad)||\mathcal{O}^{RO}(\ldots)\big)$
  - $\mathtt{State}(ID_{\mathrm{U}}, AK) \leftarrow \mathcal{O}^{RO}\big(\mathtt{State}(\mathrm{ID}_{\mathrm{C}}, AK)\big)$
  - $\mathtt{State}(ID_{\mathrm{C}}, IC) \leftarrow \mathcal{O}^{RO}\big((\mathtt{State}(ID_{\mathrm{C}}, AK) \oplus opad)||\mathcal{O}^{RO}(\ldots)\big)$
  - $\mathtt{State}(ID_{\mathrm{C}}, DC) \leftarrow \mathcal{O}^{RO}\big((\mathtt{State}(ID_{\mathrm{C}}, AK) \oplus opad)||\mathcal{O}^{RO}(\ldots)\big)$
  - $\mathtt{State}(ID_{\mathrm{C}}, AK) \leftarrow \mathcal{O}^{RO}\big(\mathtt{State}(\mathrm{ID}_{\mathrm{U}}, AK)\big)$

Since we are in the random oracle model, $Pr[S_2] = Pr[S_1]$.

Game 3: The state is only generated when the `CorruptServer` oracle is called. The IC and DC fields of the log entries are generated at random.

- `CreateUser(`λ`)`:
  - $\mathrm{U}_i \leftarrow \mathtt{pk} \leftarrow$ `GenerateKey()`
  - `DrawUser(`$\mathrm{U}_i, \mathrm{U}_i$`)` $\rightarrow vuser$

  - $\texttt{CreateEntry}(vuser, \lambda) \rightarrow l$, return U_i, l.

- $\texttt{CreateEntry}(vuser, \lambda)$:
  - $d = \texttt{Enc}_{\texttt{pk}_R}\big(\lambda, \texttt{Sign}_{\texttt{sk}_\mathrm{C}}(\lambda)\big)$ with $\texttt{pk}_R \leftarrow \texttt{GenerateKey()}$
  - $IC_\mathrm{U} \leftarrow \texttt{Rand}(|\texttt{Hash}(\cdot)|)$
  - $DC_\mathrm{U} \leftarrow \texttt{Rand}(|\texttt{Hash}(\cdot)|)$
  - $IC_\mathrm{C} \leftarrow \texttt{Rand}(|\texttt{Hash}(\cdot)|)$
  - $DC_\mathrm{C} \leftarrow \texttt{Rand}(|\texttt{Hash}(\cdot)|)$
  - $l \leftarrow \{IC_\mathrm{U}, DC_\mathrm{U}, IC_\mathrm{C}, DC_\mathrm{C}, d\}$, return l.

- $\texttt{CorruptServer}$:
  - $\forall\, \mathrm{U}_i$:
    $\texttt{State}(\mathrm{ID}_{\mathrm{U}_i}, IC) \leftarrow \texttt{Rand}(|\texttt{Hash}(\cdot)|)$
    $\texttt{State}(\mathrm{ID}_{\mathrm{U}_i}, DC) \leftarrow \texttt{Rand}(|\texttt{Hash}(\cdot)|)$
    $\texttt{State}(\mathrm{ID}_{\mathrm{U}_i}, AK) \leftarrow \texttt{Rand}(|\texttt{Hash}(\cdot)|)$
    return $\texttt{State}(\mathrm{ID}_{\mathrm{U}_i}, AK), \texttt{State}(\mathrm{ID}_{\mathrm{U}_i}, IC), \texttt{State}(\mathrm{ID}_{\mathrm{U}_i}, DC)$
  - $\texttt{State}(\mathrm{ID}_\mathrm{C}, IC) \leftarrow \texttt{Rand}(|\texttt{Hash}(\cdot)|)$
  - $\texttt{State}(\mathrm{ID}_\mathrm{C}, DC) \leftarrow \texttt{Rand}(|\texttt{Hash}(\cdot)|)$
  - $\texttt{State}(\mathrm{ID}_\mathrm{C}, AK) \leftarrow \texttt{Rand}(|\texttt{Hash}(\cdot)|)$
  - return $\texttt{State}(\mathrm{ID}_\mathrm{C}, AK), \texttt{State}(\mathrm{ID}_\mathrm{C}, IC), \texttt{State}(\mathrm{ID}_\mathrm{C}, DC)$

In this game, the adversary has no possible way to detect any relation between log entries. Hence, the probability of success is equal to guessing $Pr[S_3] = 1/2$. For this step, the output of the random oracle needs to be uniformly distributed. Hence, the corresponding input need to be different from all previous ones. In each instantiation of the $\mathcal{O}^{RO}(\cdot)$, at least part of the input is truly random or the output of the random oracle. Furthermore, the adversary has no direct input to the random oracle, the data λ is put through a randomised encryption with a randomised public key. We obtain the following bound by using the birthday paradox:

$$|Pr[S_3] - Pr[S_2]| \leq 1 - \frac{(n-1)!}{(n-1-q)!(n-1)^q}$$

with $n = 2^{|\texttt{Hash}(\cdot)|}$ and q the number of queries made to the random oracle.

Conclusion: The following advantage for an adversary against forward unlinkability of log entries is obtained for our scheme:

$$\begin{aligned}
\mathbf{Adv}^{FU}_{\mathscr{A}}(k) &= \frac{1}{2}\cdot\left|Pr\left[\mathbf{Exp}^{FU}_{\mathscr{A}}(k)=1|b=0\right]+\right.\\
&\quad \left.Pr\left[\mathbf{Exp}^{FU}_{\mathscr{A}}(k)=1|b=1\right]-1\right|\\
&= |Pr[S_0]-Pr[S_3]|\\
&= |Pr[S_0]-Pr[S_1]+Pr[S_2]-Pr[S_3]|\\
&\leq |Pr[S_0]-Pr[S_1]|+|Pr[S_2]-Pr[S_3]|\\
&\leq \mathbf{Adv}^{DDH}(k)+1-\frac{(n-1)!}{(n-1-q)!(n-1)^q}.
\end{aligned}$$

In the random oracle model, under the DDH assumption, this advantage is negligible. □

3.5.4 Unlinkability of User Identifiers

Theorem 5. *Under the DDH assumption, the proposed logging scheme provides computational forward unlinkability of user identifiers, according to Definition 5.*

Proof. First, we will show that the distribution of the randomised user identifiers is equal to the distribution of user identifiers, uniformly at random. User identifiers are generated by $\{\mathtt{sk},\mathtt{pk}\} \leftarrow \mathtt{GenerateKey()}$, where $\mathtt{sk} = x$ is chosen uniformly at random modulo n and $\mathtt{pk} = X = xG$, for G a generator of the underlying curve. A blinded key is generated as follows: $X' = X + rG = (x+r)G$, with r uniformly chosen at random modulo n. The distribution of $(x+r)$ is also uniformly at random modulo n, since x and r are independent.

Second, we will show by reduction that there exists no adversary against the unlinkability of user identifiers. Assume an adversary $\mathscr{A}$ with non-negligible probability of success in determining whether or not $\langle \mathrm{ID}_U, \mathrm{ID}'_U, d \leftarrow l\rangle$ is a valid tuple, i.e., of the form $\langle x_1 G, x_2 G, \mathtt{Enc}_{X_1}(M_f, x_2 - x_1, \ldots)\rangle$. The used encryption scheme is ECIES, for which the ciphertext (R, c, m) of a plaintext p under public key X is computed as follows:

$$R = rG \quad c = \mathrm{ENC}_{K_0}(p) \quad m = \mathtt{MAC}_{K_1}(c), \text{ with } r \in_R \mathbb{Z}_n \text{ and}$$

$$K_0 || K_1 = KDF\big(xcoord(rX)\big).$$

Given this adversary $\mathscr{A}$, another adversary can be constructed against an instance of the DDH problem: given $\langle A = aG, B = bG, D\rangle$, determine whether or not $D = abG$. For a valid DDH tuple, $\mathscr{A}$ will have a non-negligible advantage when given the following: for $t \in_R \mathbb{Z}_n$, $\langle A, A + tG, \mathtt{Enc}_A(M_f, t, \dots)\rangle$, where for the encryption we use $rG = B$ and $K_0 || K_1 = KDF(xcoord(D))$. If the DDH tuple was not valid, $\mathscr{A}$ can only guess and has probability of 1/2 to win the game. □

3.6 Performance

We implemented a prototype of our scheme in the programming language Go[5], with the cryptographic primitives ECIES and ECDSA on NIST P-256, SHA-256, and HMAC using SHA-256. All benchmarks are performed on a laptop running a GNU/Linux based x64 OS with an Intel i5 (quad core 2.6GHz) CPU and 7.7 GB DDR3 RAM. Table 3.1 provides a benchmark of the algorithms and protocols that make up our scheme, done by Go's built-in benchmarking functionality, which will run a test until it is "timed reliably". The benchmark shows that operations related to encryption and signatures are the main bottlenecks. As a consequence, controllers perform the bulk of the work. Decryption and verification for users are relatively costly. However, in practice, downloading the log entries over an anonymous channel (such as that provided by Tor [45]) possibly with an imposed waiting time, as discussed in Section 3.4.6, will most likely cause the bottleneck. For servers, creating log entries is fast, while setup (also part of forking) is costly. Presumably, setup will be relatively infrequent.

Table 3.1 Benchmark of algorithms and protocols.

Algorithm/Protocol	Time [ms]	Comments
Algorithm 1: Setup Entity	14,8	
Protocol 1: Setup U,S,C	25,0	
Algorithm 2: Create Log Entry	0,1	1 KiB data
Protocol 2: Generate Log Entry	15,0	1 KiB data
Protocol 3: Forking	45,3	
Protocol 4: Verify	180,2	10 entries of 1 KiB data

[5] https://golang.org, version 1.1 RC2, accessed 2013-02-14.

To get a better idea of how our scheme would perform in practice as a deployed system, we extended our implementation. First, we transformed the controller to a standalone service (similar to a syslog server) to which other systems at the controller send messages that should be logged. The controller and server provide RESTful APIs over HTTPS. TLS is provided with the cipher suite ECDHE/RSA/AES/256/CBC/SHA, where the 2048-bit RSA keys are pre-shared. Next, we introduced the concept of *transactions*, analogous to transactions in relational databases. At a controller, starting a transaction creates a new buffer for user-message pairs. A transaction can then be committed, which takes all user-message pairs in the buffer and creates log entries of them as described in our scheme. At a server, a transaction buffer works in a similar way: a controller can create a buffer of index-user-message triplets that when committed gets added as log entries at the server, in the order of their indices. This index is needed because our scheme requires that the order of log entries is preserved. In return, this means that the controller can easily send index-user-message triplets in parallel to the server. For the transaction buffers at a controller we also added support for parallelism. When a controller has received a user-message pair, the controller spawns a new Go routine (similar to a lightweight thread) that performs the signing and encryption of the message in the background. This allows the controller to quickly let other systems at the controller return to their primary work of data processing, and the controller service can perform the heavy computations while waiting for the transaction to be committed.

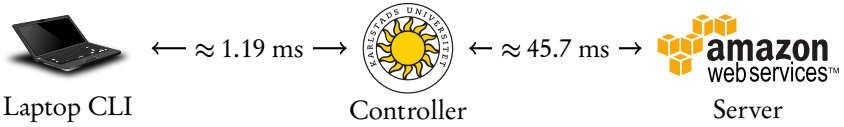
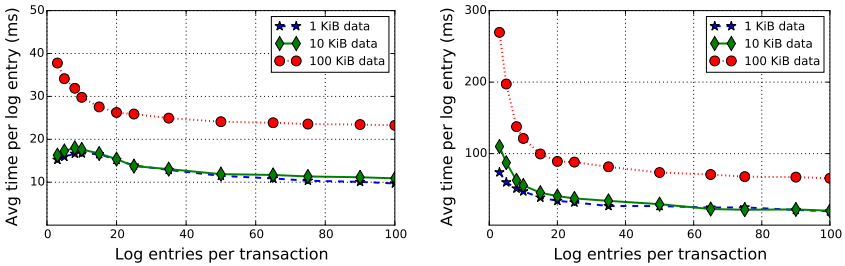


Figure 3.5 The setting of our remote experiment.

We conducted two experiments to evaluate the performance of our scheme. The server, the controller, and the application calling the controller API were run both locally and remotely. The local experiment was run on the previously described laptop. For the remote experiment, the server was run at Amazon and the controller in a private cloud at Karlstad University in Sweden. Figure 3.5 illustrates the setting of our remote experiment. The application calling the controller API was run on the same laptop as used previously connected to the same university network as the controller. The Amazon EC2 instance was an M1 Medium instance with 2 ECUs (EC2 Compute Unit), 1 core, and 3.7 GiB of memory. It was hosted in the

eu-west-1c zone and ran Ubuntu GNU/Linux 12.10 x64. The private cloud instance was a VMware Ubuntu GNU/Linux 12.04 x64 instance with access to a quad core Intel Xeon E5540[6] CPU and 4 GiB of memory. We used the Linux *ping* command to measure the latency between 1) the controller and the Amazon EC2 instance: on average 45.7 ms with a standard deviation of 0.3 ms, and 2) the laptop and the controller: on average 1.19 ms with a standard deviation of 0.09 ms. Figure 3.6 shows the average time (in milliseconds) for generating a log entry depending on the number of entries in each transaction and the size of the data to be logged for each entry. Each entry was logged for a random user that was set up before the start of the experiment.



(a) Local setting. **(b)** Remote setting.

Figure 3.6 The average time (ms) for generating a log entry depending on the number of log entries per transaction and the size of the data to be logged.

The result depicted in Figure 3.6 clearly shows that a modest transaction size significantly lowers the average log entry generation time, with a more clear gain in the remote setting. At around 20 entries per transaction, the benefit of adding more entries to a transaction diminishes. The average log entry time does not scale linearly with the size of the logged data. This is mainly due to the fact that the data to be logged is first signed and then encrypted before being sent to the server. The relative overhead for the signature and public-key encryption is larger for smaller log entries. In the local setting, the difference between generating a log entry with 1 KiB and 100 KiB of data is roughly 13 ms at 50 entries per transaction, resulting in about double the average log entry generation time for logging 100 times more data. In the remote setting, logging 100 KiB of data per entry takes roughly 3 times as long as logging 1 KiB of data. We suspect the increased time in the remote setting

[6]Note that Intel Xeon E5540 does not support AES-NI by default, see http://ark.intel.com/products/37104/Intel-Xeon-Processor-E5540-(8M-Cache-2_53-GHz-5_86-GTs-Intel-QPI), accessed 2015-03-31.

is due to the increased latency, but further experiments are needed. Table 3.2 shows the *goodput*, i.e., the throughput measured with respect to the data to be logged, for both the local and remote setting at 100 log entries per transaction.

Table 3.2 Goodput at 100 log entries per transaction.

Goodput (KiB/s)	Data per entry		
	1 KiB	10 KiB	100 KiB
Local setting	87	842	4149
Remote setting	52	497	1525

3.7 Conclusions

We introduced the first provably secure privacy-preserving transparency logging scheme, supporting distributed log trails. Dynamic and distributed processes can be logged, and the logging itself can also be distributed across several servers. The log entries are world-readable, but the strong cryptographic properties of the underlying scheme ensure confidentiality and unlinkability in a broad sense. Moreover, we are the first to formalise these properties in a general way, enabling us to prove that our scheme meets the necessary requirements. We also implemented our scheme in a robust prototype implementation and the first timing results show that the tool can be used in practice.

Returning to the 15 requirements from the previous chapter in Section 2.5 on page 34, we briefly go over how well the scheme from this chapter addresses them. Requirements R1–R3 are covered, in part thanks to the secrecy and unlinkability properties of the scheme, restricting reconstruction and access to the contents to users only. The deletion-detection forward integrity property fulfils the R4 requirement for users, and similarly, the controller's index chain and data chain checks R5 and R6 for controllers. R7 is only possible if controllers provide their initial authentication keys. The first three privacy requirements, R8–R10, are covered by our forward unlinkability of entries property. The last privacy requirement, R11, is fulfilled by the forward unlinkability of user identifiers property. The auditability and accountability properties are more problematic. R12 is partly solved, within the adversary model, by the data chain for the controller as part of each entry. R13 is partly covered by the signature on all data and the user chain. R14 and R15 are not covered by our scheme

as presented in this chapter. In early work, we sketched an auditing protocol between a controller and server [97], involving signatures on the current controller data chain and periodic time-stamping. This protocol was never thoroughly evaluated though. Next, we reflect on the limitations of our work so far and explore possible avenues for moving forward, including how to approach the auditing and accountability requirements.

Chapter 4

Limitations and Hardware

> It is not just the right of the person who speaks to be heard, it is the right of everyone in the audience to listen and to hear, and every time you silence somebody you make yourself a prisoner of your own action because you deny yourself the right to hear something. In other words, your own right to hear and be exposed is as much involved in all these cases as is the right of the other to voice his or her view.
>
> *Christopher Hitchens*

In this chapter, we focus on a number of limitations of the scheme proposed in the previous chapter. After identifying the limitations, we present a potential solution to some of the limitations in the form of relying on trusted hardware. Reflecting on this potential solution, we motivate the decision moving forward to focus on authenticated data structures.

4.1 Limitations

The following six subsections each describe a key limitation of the scheme presented in the previous chapter.

4.1.1 Only Private Verifiability

Each user can individually verify his or her log entries, ensuring that no entry has been modified or deleted (within the adversary model), due to our scheme providing deletion-detection forward integrity. Similarly, the controller has the same capability for *all* log entries it produces. This comes at the cost of each log entry having an IC and DC field dedicated to the controller, and the server keeping state for the controller. For auditing purposes (identified as requirements R12–R15 in Chapter 2), towards an auditor (or any other third party), this private verifiability approach becomes problematic. Using the DC field of log entries to convince an auditor involves revealing authentication keys, which, e.g., gives the auditor the ability to recompute the DC field values. Ideally, the fact that no entries have been modified or deleted should be *publicly verifiable*, such that no secrets are required.

4.1.2 One Entry Per Insert

Every time the controller wants to create a new log entry it first signs and encrypts the data, then sends the resulting ciphertext to the server that creates the log entry. We added support for transactions, which provides the ability to insert a set of log entries when the transaction is committed. Transactions generate extra round trips though, from the caller of the controller API to the controller API and from the controller to the server, for creating and committing transactions. While this enables the use of the scheme in such a way that any implementation issues may not leak the order entries are inserted, as discussed in Chapter 2, it comes at the cost of extra overhead and shifts complexity to the system designer wishing to perform transparency logging. Ideally, set insertion of entries should be the default and promote safe use without incurring extra overhead.

4.1.3 Limited Controller Verifiability

While not strictly a problem within our adversary model, where *all* controllers and servers are collectively assumed forward-secure, a practical problem is the limited controller verifiability of log entries created by the server. To create a log entry, the

controller first signs a message and then encrypts the message and resulting signature under the public key of a user. The ciphertext is sent to the server together with the identity (public key) of the user, such that the server knows which entry in its state it should use to create the two user-fields (IC and DC). Herein lies the gist of the limited controller verifiability: a malicious (already compromised) server can create the entry for any user (or none), and there is no way for the controller to verify which user the entry is created for. The controller is limited to verifying the controller-fields (IC and DC) of an entry, and that they cover the ciphertext it produced. Conversely, a malicious controller can instruct an honest server to create an entry for the wrong user. We consider this case less relevant, since a malicious controller will presumably not provide an honest account of its processing. Ideally, a server should not have to be trusted and still be able to perform its role.

4.1.4 Unreasonable Outsourcing of State

The scheme presented in Chapter 3 outsources keeping state for creating entries and the storage of the entries to servers. While outsourcing the storage of entries removes the burden of storage and interaction with users when the users request entries, the outsourcing of state is more problematic. It is true that keeping state, especially with the goal of providing forward security, is a practical burden, but completely outsourcing state leads to problems like limited controller verifiability. Also, for sake of enabling an audit by a third-party, we also sketched an auditing protocol between controllers and servers in our early work to partly solve this issue (not presented in this dissertation), that resulted in some (although constant sized) state at the controller [97]. This state also had to be forward-secure, because otherwise an adversary could use the controller IC or DC field to order all entries chronologically. If evolving state is to play a role, outsourcing it to servers cause bigger problems than it solves.

4.1.5 Several Forward-Secure Entities

As previously mentioned, Chapter 3 modelled both controllers and servers as forward secure. Having to place that level of trust into servers is the source of many problems, such as the limited controller verifiability problem and the issues around outsourcing state. Such a strong level of trust also rules out using commodity cloud services due to the inherent security and privacy risks. Ideally, the server should be untrusted. This would also make more sense in the transparency logging setting, since the focus is on

controllers and users. Servers are secondary entities. Having to assume at minimum forward-secure controllers may be inevitable, since they produce the descriptions of their own data processing, but servers only take on some of the work associated to transparency logging from controllers.

4.1.6 Unsafe Primitives

The scheme from Chapter 3 selected, among others, ECDSA for signatures and ECIES for encryption over the NIST P-256 elliptic curve as instantiated primitives. The NIST P-256 curve, while a "safe" choice in the sense of being standardised and available in common cryptographic libraries, its "safety" in the sense of design, ease of implementation and efficiency is questionable[1]. Future work should use safer options, such as Curve25519 [22], and higher-level (in terms of abstraction) libraries like NaCl [25], to not shift the burden of correctly implementing low-level cryptographic operations to implementers.

4.2 A Potential Solution: Trusted Hardware

Here we present work done in parallel to the design of the scheme presented in Chapter 3 on using trusted hardware to reduce the necessary trust in servers. The goal of using hardware is to create a *trusted state*, where the server only acts as an intermediary between the controller and the hardware, and provides storage of the generated entries. An additional perk is that the hardware can, similar to a hardware security module (HSM), keep the server's signing key secured and potentially speed up cryptographic operations. As part of this, we sketched the following five algorithms to be provided by the hardware towards the server:

SetupController In a controlled environment, where the controller can be assured that it is interacting with the hardware board, state is setup for the controller. This mimics Algorithm 1 on page 60.

SetupUser To setup a user, the controller symmetrically encrypts the public key of the user using the current authentication key for the controller. This mimics Algorithm 1 on page 60, with the difference from SetupController that this only can setup a user and the user's public key is encrypted.

[1] See http://safecurves.cr.yp.to/ by Daniel J. Bernstein and Tanja Lange. SafeCurves: choosing safe curves for elliptic-curve cryptography. Accessed 2015-04-06.

CreateEntry Provided the symmetrically encrypted public key of the user and the hash of the entry ciphertext, the hardware creates the entry as in Algorithm 2 on page 62. The server stores the entry together with the actual ciphertext.

RemoveUser Provided the symmetrically encrypted public key of the user, removes the user from state.

GetState Provided the public key of a user, returns state, as in Protocol 4 on page 64. No changes.

The basic idea behind the changes is to use the shared secret (current authentication key) between the controller and hardware as an additional layer of protection against the server. The goal of our work is to evaluate the feasibility of putting state in hardware before working out the exact algorithms.

We built a proof-of-concept implementation on a Field Programmable Gate Array (FPGA), the Xilinx Virtex-5 FPGA. Figure 4.1 shows the high-level architecture of the trusted state component on the FPGA. The MicroBlaze softcore processor from Xilinx is the central processing unit (CPU) that relays the communication from the server to the trusted state component. The processor local bus (PLB) connects the MicroBlaze CPU with the trusted state component and the Ethernet core. The Ethernet core enables communication with the MicroBlaze CPU over Ethernet, where the MicroBlaze CPU adds support for the UDP/IP protocol. The MicroBlaze CPU communicates with the trusted state component over the PLB bus using a command register (CR) and a status register (SR). In the trusted state component, the five internal components (ECP, HASH, RNG, SYM. KEY CIPHER, and the memory component) are interconnected over shared block RAM (BRAM). A simple finite state machine (FSM) coordinates the components. The memory component is stored on volatile BRAM on the FPGA (commonly found on the die of a FPGA). The cryptographic components use designs focused on area not speed. The four cryptographic components shown in Figure 4.1 are:

ECP An elliptic-curve processor by Vliegen *et al.* [117] on NIST P-256 used for ECDSA and ECIES.

HASH An implementation of the SHA-256 hash function by Rykx and Thielen [102] for hashing and HMAC.

RNG A random number generator by Wold and Tan [121] for ECDSA, ECIES, and generating initial authentication keys.

SYM. KEY CIPHER An implementation of AES-128 by Chodowiec and Gaj [34].

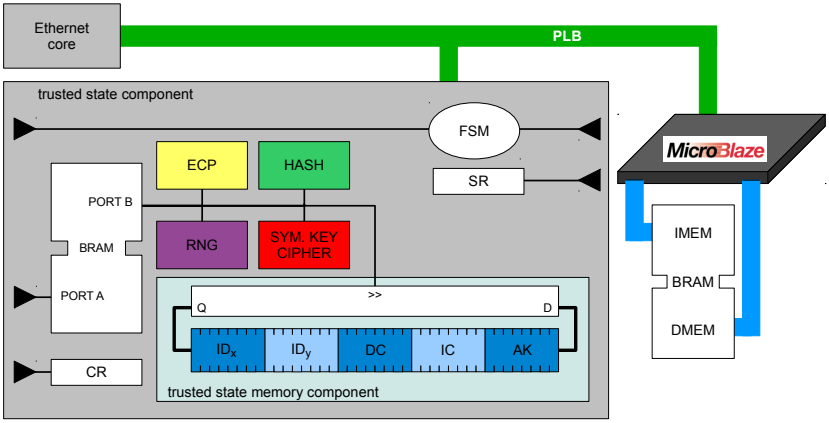


Figure 4.1 The high-level architecture of the trusted state component on the FPGA.

Table 4.1 shows a quick micro-benchmark of the five algorithms for the trusted state hardware and a pure software implementation in Java. The GetState algorithm was not thoroughly evaluated in hardware. The software implementation simulates the hardware in software running on a commodity laptop. The software benchmarks measure the algorithm directly using a testing framework, while the hardware measurements are done by counting clock cycles. For the hardware, the running time (ms) for target 0 and 511 are shown. The target is the location in BRAM of the state entry being processed. The higher the target, the longer it takes to access the memory location, since the implementation requires BRAM to be searched sequentially until the target state is found. For the hardware, the impact of small and big scalars are shown, as it influences the number of operations on the elliptic curve the co-processor has to perform. The hardware implementation shows worse performance than the software implementation in general, especially for state entries with a high target in BRAM and big scalars. Since random scalars are generated as part of ECIES and ECDSA, the expected average scalar is somewhere between small and big in the table. A more detailed analysis, presented in [97, 116, 118], reveals that the memory access, size of the elliptic curve scalars, and hash function are the main sources of poor hardware performance. We note that the cryptographic components were selected for minimum area, not speed.

Table 4.1 A micro-benchmark of four of the algorithms for trusted state, comparing the hardware implementation to a software implementation in Java.

Running time (ms)	Hardware				Software
	Target at 0		Target at 511		
	small scalar	big scalar	small scalar	big scalar	
SetupController	68.47	167.10	79.66	178.29	59
SetupUser	68.72	167.35	91.10	189.73	60
CreateEntry	1.07		23.45		0.062
RemoveUser	9.71		20.90		0.020
GetState	-	-	-	-	41

Hardware in hostile environments is hard to secure. The full life-cycle, from design and construction to supply chain has to be taken into account before the hardware can even start to be used. During use, ensuring that the hardware is tamper proof, even against various side-channel attacks like timing and power analysis, are potentially even harder [83]. Hardware is also costly and operational pains like keeping backups and providing redundancy does not ease deployment.

4.3 Moving Forward

Hardware offers a solution to some of the identified limitations, most notably reducing the necessary trust in the server, but also to a degree may provide faster primitives (at least with the right components). This comes at the added complexity of having to trust a piece of hardware in a hostile environment, which is non-trivial. In a sense, introducing a trusted state in hardware is a solution to a problem with trust by shifting the trust from one entity to another. The hardware in no way addresses private verifiability and solidifies the outsourcing of state that adds complexity to the scheme. With proper definitions from Chapter 3 on several key security and privacy properties, we opted to instead of further investigating hardware to look for solutions that would enable us to treat the server as completely untrusted. We found a promising solution in *authenticated data structures*.

Chapter 5

Balloon: An Append-Only Authenticated Data Structure

> All aspects of government work best when the relative power between the governors and the governed remains as small as possible – when liberty is high and control is low. Forced openness in government reduces the relative power differential between the two, and is generally good. Forced openness in laypeople increases the relative power, and is generally bad.
>
> *Bruce Schneier* – The Myth of the 'Transparent Society'

This chapter presents a novel *append-only authenticated data structure* that removes the need to trust the server in transparency logging without the need for trusted hardware. Our data structure, which is called Balloon[1], allows for efficient proofs of both membership and non-membership of entries. As such, the server is forced to

[1] Like an ideal balloon, a vacuum balloon, our balloon is filled with nothingness (seemingly random data) without giving in to any external pressures (attackers) to collapse (remove or replace nothingness with something).

provide a verifiable reply to all membership queries. Balloon also provides efficient (non-)membership proofs for past versions of the data structure (making it *persistent*), which is a key property for providing proofs of time, when only some versions of the Balloon have been timestamped. Since Balloon is append-only, we can greatly improve the efficiency in comparison with other authenticated data structures, such as persistent authenticated dictionaries [8].

We take a similar approach as Papamanthou *et al.* [86] for formally defining and proving the security of Balloon. We view Balloon in the model of *authenticated data structures* (ADS), using the *three-party* setting [112]. The three party setting for ADS consists of the *source* (corresponding to our controller), one or more *servers*, and one or more *clients* (corresponding to our users). The source is a trusted party that controls a data structure (the Balloon) that is copied to the untrusted servers together with some additional data that authenticates the data structure. The servers answer queries made by clients. The goal for an ADS is for clients to be able to verify the correctness of the queries based only on public information. The public information takes the form of a verification key, for verifying signatures made by the source, and some *digest* produced by the source to authenticate the data structure. The source can update the ADS, in the process producing new digests. In our case, we refer to the digests as *snapshots*. The query we want to enable clients to verify is a *membership query*, which proves *membership* or *non-membership* of an entry with a provided key for a provided snapshot.

After we show that Balloon is a secure ADS in the three party setting, we extend Balloon to enable the controller (source) to not keep a copy of the data structure and still be able to perform verifiable inserts of new entries to update the Balloon. Finally, we describe how *monitors* and a *perfect gossiping mechanism* would prevent the controller from undetectably modifying or deleting entries once inserted into the Balloon, which lays the foundation for the forward-secure controller setting.

The chapter is structured as follows. Section 5.1 introduces the background of our idea. Section 5.2 defined Balloon as an ADS that allows for both efficient membership and non-membership proofs, also for past versions of the Balloon, while keeping the storage and memory requirements minimal. We formally prove that Balloon is a correct and secure authenticated data structure in Section 5.3 according to the definitions by Papamanthou *et al.* [86]. Section 5.4 describes how to produce efficient verifiable inserts into our append-only authenticated data structure that enable the controller to ensure consistency of the data structure without storing a copy of the entire (authenticated) data structure. Section 5.5 defines publicly verifiable consistency

for an ADS scheme and shows how it enables a forward-secure source. Section 5.6 shows that Balloon is applicable in practice, providing performance results for a proof-of-concept implementation. Section 5.7 presents related work and compares Balloon to prior work. Section 5.8 shows why probabilistic proofs are insufficient for ensuring consistency of a Balloon without greatly increasing the burden on the server. Section 5.9 concludes the chapter.

5.1 Preliminaries

First, we introduce the used formalisation of an authenticated data structure scheme. Next, we give some background on the two data structures that make up Balloon: a history tree, for efficient membership proofs for any snapshot, and a hash treap, for efficient non-membership proofs. Finally we present our cryptographic building blocks.

5.1.1 An Authenticated Data Structure Scheme

Papamanthou *et al.* [86] define an authenticated data structure and its two main properties: correctness and security. We make use of these definitions and therefore present them here, be it with slight modifications to fit our terminology.

Definition 6 (ADS scheme)**.** *Let D be any data structure that supports queries q and updates u. Let* $\mathtt{auth}(D)$ *denote the resulting authenticated data structure and* $\mathtt{s}$ *the snapshot of the authenticated data structure, i.e., a constant-size description of D. An ADS scheme $\mathcal{D}$ is a collection of the following six probabilistic polynomial-time algorithms:*

1. $\{\mathtt{sk},\mathtt{pk}\} \leftarrow \mathtt{genkey}(1^\lambda)$*: On input the security parameter λ, it outputs a secret key* $\mathtt{sk}$ *and public key* $\mathtt{pk}$*;*

2. $\{\mathtt{auth}(D_0),\mathtt{s}_0\} \leftarrow \mathtt{setup}(D_0,\mathtt{sk},\mathtt{pk})$*: On input a (plain) data structure D_0, the secret key* $\mathtt{sk}$*, and the public key* $\mathtt{pk}$*, it computes the authenticated data structure* $\mathtt{auth}(D_0)$ *and the corresponding snapshot* $\mathtt{s}_0$*;*

3. $\{D_{b+1},\mathtt{auth}(D_{b+1}),\mathtt{s}_{b+1},\mathtt{upd}\} \leftarrow \mathtt{update}(u,D_b,\mathtt{auth}(D_b),\mathtt{s}_b,\mathtt{sk},\mathtt{pk})$*: On input an update u on the data structure D_b, the authenticated data structure* $\mathtt{auth}(D_b)$*, the snapshot* $\mathtt{s}_b$*, the secret key* $\mathtt{sk}$*, and the public key* $\mathtt{pk}$*, it outputs the updated data structure D_{b+1} along with the updated authenticated data structure* $\mathtt{auth}(D_{b+1})$*, the updated snapshot* $\mathtt{s}_{b+1}$ *and some relative information* $\mathtt{upd}$*;*

4. $\{D_{h+1}, \texttt{auth}(D_{h+1}), \texttt{s}_{h+1}\} \leftarrow \texttt{refresh}(u, D_h, \texttt{auth}(D_h), \texttt{s}_h, \texttt{upd}, \texttt{pk})$*: On input an update u on the data structure D_h, the authenticated data structure* $\texttt{auth}(D_h)$*, the snapshot* $\texttt{s}_h$*, relative information* upd *and the public key* pk*, it outputs the updated data structure D_{h+1} along with the updated authenticated data structure* $\texttt{auth}(D_{h+1})$ *and the updated snapshot* $\texttt{s}_{h+1}$*;*

5. $\{\Pi(q), \alpha(q)\} \leftarrow \texttt{query}(q, D_h, \texttt{auth}(D_h), \texttt{pk})$*: On input a query q on data structure D_h, the authenticated data structure* $\texttt{auth}(D_h)$ *and the public key* pk*, it returns the answer $\alpha(q)$ to the query, along with proof $\Pi(q)$;*

6. $\{\texttt{accept}, \texttt{reject}\} \leftarrow \texttt{verify}(q, \alpha, \Pi, \texttt{s}_h, \texttt{pk})$*: On input query q, an answer α, a proof Π, a snapshot* $\texttt{s}_h$ *and the public key* pk*, it outputs either* accept *or* reject*.*

Next to the definition of the ADS scheme, another algorithm was defined for deciding whether or not an answer α to query q on data structure D_h is correct: $\{\texttt{accept}, \texttt{reject}\} \leftarrow \texttt{check}(q, \alpha, D_h)$.

Definition 7 (Correctness)**.** *Let Balloon be an ADS scheme {*genkey,setup, update, refresh,query,verify*}. We say that the ADS scheme Balloon is correct if, for all $\lambda \in \mathbb{N}$, for all* $\{\texttt{sk}, \texttt{pk}\}$ *output by algorithm* genkey*, for all D_h,* $\texttt{auth}(D_h)$*,* $\texttt{s}_h$ *output by one invocation of* setup *followed by polynomially-many invocations of* refresh*, where $h \geq 0$, for all queries q and for all $\Pi(q), \alpha(q)$ output by* $\texttt{query}(q, D_h, \texttt{auth}(D_h), \texttt{pk})$ *with all but negligible probability, whenever algorithm* $\texttt{check}(q, \alpha(q), D_h)$ *outputs* accept*, so does* $\texttt{verify}(q, \Pi(q), \alpha(q), \texttt{s}_h, \texttt{pk})$.

Definition 8 (Security)**.** *Let Balloon be an ADS scheme {*genkey,setup, update, refresh,query,verify*}, λ be the security parameter, $\epsilon(\lambda)$ be a negligible function and* $\{\texttt{sk}, \texttt{pk}\} \leftarrow \texttt{genkey}(1^\lambda)$*. Let also* **Adv** *be a probabilistic polynomial-time adversary that is only given* pk*. The adversary has unlimited access to all algorithms of Balloon, except for algorithms* setup *and* update *to which he has only oracle access. The adversary picks an initial state of the data structure D_0 and computes* $D_0, \texttt{auth}(D_0), \texttt{s}_0$ *through oracle access to algorithm* setup*. Then, for $i = 0, ..., h = \texttt{poly}(\lambda)$,* **Adv** *issues an update u_i in the data structure D_i and computes* $D_{i+1}, \texttt{auth}(D_{i+1})$ *and* $\texttt{s}_{i+1}$ *through oracle access to algorithm* update*. Finally the adversary picks an index $0 \leq t \leq h+1$, and computes a query q, answer α and a proof Π. We say that the ADS scheme Balloon is secure if for all $\lambda \in \mathbb{N}$, for all* $\{\texttt{sk}, \texttt{pk}\}$ *output by algorithm* genkey*, and for any probabilistic*

polynomial-time adversary **Adv** *it holds that*

$$Pr\left[\begin{array}{r}\{q,\Pi,\alpha,t\}\leftarrow \mathbf{Adv}(1^{\lambda},\mathtt{pk});\mathtt{accept}\leftarrow \mathtt{verify}(q,\alpha,\Pi,\mathtt{s}_t,\mathtt{pk})\\ \mathtt{reject}\leftarrow \mathtt{check}(q,\alpha,D_t).\end{array}\right]\leq \epsilon(\lambda) \quad (5.1)$$

5.1.2 History Tree

A tamper-evident history system, as defined by Crosby and Wallach [38], consists of a *history tree* data structure and five algorithms. A history tree is in essence a *versioned* Merkle tree [79] (hash tree). Each leaf node in the tree is the hash of an entry, while interior nodes are labeled with the hash of its children nodes in the subtree rooted at that node. The root of the tree fixes the content of the entire tree. Different versions of history trees, produced as entries are added, can be proven to make consistent claims about the past. The five algorithms, adjusted to our terminology, were defined as follows:

- $c_i \leftarrow \mathtt{H.Add}(e)$: Given an entry e the system appends it to the history tree H as the i:th entry and then outputs a commitment[2] c_i.

- $\{P, e_i\} \leftarrow \mathtt{H.MembershipGen}(i, c_j)$: Generates a membership proof P for the i:th entry with respect to commitment c_j, where $i \leq j$, from the history tree H. The algorithm outputs P and the entry e_i.

- $P \leftarrow \mathtt{H.IncGen}(c_i, c_j)$: Generates an incremental proof P between c_i and c_j, where $i \leq j$, from the history tree H. Outputs P.

- $\{\mathtt{accept},\mathtt{reject}\} \leftarrow \mathtt{P.MembershipVerify}(i, c_j, e_i')$: Verifies that P proves that e_i' is the i:th entry in the history fixed by c_j (where $i \leq j$). Outputs `accept` if true, otherwise `reject`.

- $\{\mathtt{accept},\mathtt{reject}\} \leftarrow \mathtt{P.IncVerify}(c_i', c_j)$: Verifies that P proves that c_j fixes every entry fixed by c_i' (where $i \leq j$). Outputs `accept` if true, otherwise `reject`.

[2] A commitment c_i is the root of the history tree for the i:th entry, signed by the system. For the purpose of our work, we omit the signature from the commitments.

5.1.3 Hash Treap

A treap is a type of randomised binary search tree [13], where the binary search tree is balanced using heap priorities. Each node in a treap has a key, value, priority, and a left and a right child. A treap has three important properties:

1. Traversing the treap in order gives the sorted order of the keys.
2. Treaps are structured according to the nodes' priorities, where each node's children have lower priorities.
3. Given a deterministic attribution of priorities to nodes, a treap is *set unique* and *history independent*, i.e., its structure is unique for a given set of nodes, regardless of the order in which nodes were inserted, and the structure does not leak any information about the order in which nodes were inserted.

When a node is inserted in a treap, its position in the treap is first determined by a binary search. Once the position is found, the node is inserted in place, and then rotated upwards towards the root until its priority is consistent with the heap priority. When the priorities are assigned to nodes using a cryptographic hash function, the tree becomes probabilistically balanced with an expected depth of $\log n$, where n is the number of nodes in the treap. Inserting a node takes expected $\mathcal{O}(\log n)$ operations and results in expected $\mathcal{O}(1)$ rotations to preserve the properties of the treap [37]. Given a treap, it is straightforward to build a hash treap: have each node calculate the hash of its own attributes[3] together with the hash of its children. Since the hash treap is a Merkle tree, its root fixes the entire hash treap. The concept of turning treaps into Merkle trees for authenticating the treap has been used for example in the context of persistent authenticated dictionaries [39] and authentication of certificate revocation lists [84].

We define the following algorithms on our hash treap, for which we assume that keys k are unique and of predefined constant size $\texttt{cst}$:

- $r \leftarrow \texttt{T.Add}(k, v)$: Given a unique key k and value v, where $|k| = \texttt{cst}$ and $|v| > 0$, the system inserts them into the hash treap T and then outputs the updated hash of the root r. The add is done with priority $\texttt{Hash}(k)$, which results in a deterministic treap. After the new node is in place, the hash of each node along the path from the root has its internal hash updated. The hash of

[3] The priority can safely be discarded since it is derived solely from the key and implicit in the structure of the treap.

a node is $\texttt{Hash}(k||v||\text{left.hash}||\text{right.hash})$. In case there is no right (left) child node, the right.hash (left.hash) is set to a string of consecutive zeros of size equal to the output of the used hash function $0^{|\texttt{Hash}(\cdot)|}$.

- $\{P^T, v\} \leftarrow \texttt{T.AuthPath}(k)$: Generates an authenticated path P^T from the root of the treap T to the key k where $|k| = \texttt{cst}$. The algorithm outputs P^T and, in case of when a node with key k was found, the associated value v. For each node i in P^T, k_i and v_i need to be provided to verify the hash in the authenticated path.

- $\{\texttt{accept}, \texttt{reject}\} \leftarrow P^T.\texttt{AuthPathVerify}(k, v)$: Verifies that P^T proves that k is a non-member if $v \stackrel{?}{=} \texttt{null}$ or otherwise a member. Verification checks that $|k| = \texttt{cst}$ and $|v| > 0$ (if $\neq \texttt{null}$), calculates and compares the authenticator for each node in P^T, and checks that each node in P^T adheres to the sorted order of keys and heap priority.

Additionally we define the following helper algorithm for the hash treap:

- $\texttt{pruned(T)} \leftarrow \texttt{BuildPrunedTree}(< P_j^T >)$: Generates a pruned hash treap $\texttt{pruned(T)}$ from the given authenticated paths P_j^T in the hash treap T. This algorithm removes any redundancy between the authenticated paths, resulting in a more compact representation as a pruned hash treap. Note that evaluating $\texttt{pruned(T).AuthPathVerify}(k, v)$ is equivalent with evaluating $P^T.\texttt{AuthPathVerify}(k, v)$ on the authenticated path P^T through k contained in the pruned hash treap.

- $r \leftarrow P^T.\texttt{root}()$: Outputs the root of the authenticated path. Note that $\texttt{pruned(T).root}()$ and $P^T.\texttt{root}()$ are equivalent for any authenticated path P^T contained by the pruned tree.

5.1.4 Cryptographic Building Blocks

We assume idealised cryptographic building blocks in the form of a hash function $\texttt{Hash}(\cdot)$, and signature scheme that is used to sign a message m and verify the resulting signature: $\{\texttt{accept}, \texttt{reject}\} \leftarrow \texttt{Verify}_{\texttt{vk}}(\texttt{Sign}_{\texttt{sk}}(m), m)$. The hash function should be collision and pre-image resistant. The signature scheme should be existentially unforgeable under known message attack. Furthermore, we rely on the following lemma for the correctness and security of a Balloon:

Lemma 1. *The security of an authenticated path in a Merkle (hash) tree reduces to the collision resistance of the underlying hash function.*

Proof. This follows from the work by Merkle [80] and Blum *et al.* [27]. □

5.2 Construction and Algorithms

Our data structure is an append-only key-value store that stores entries e consisting of a key k and a value v. Each key k_i is assumed to be unique and of predefined constant size `cst`, where `cst` $\leftarrow |\texttt{Hash}(\cdot)|$. Additionally, our data structure encodes some extra information in order to identify in which set (epoch) entries were added. We define the algorithm $k \leftarrow \texttt{key}(e)$ that returns the key k of the entry e.

Our authenticated data structure combines a hash treap and a history tree when adding an entry e as follows:

- First, the entry is added to the history tree: $c_i \leftarrow H.add\big(\texttt{Hash}(k||v)\big)$. Let i be the index where the hashed entry was inserted at into the history tree.
- Next, the hash of the entry key $\texttt{Hash}(k)$ and the entry position i are added to the hash treap: $r \leftarrow \texttt{T.Add}(\texttt{Hash}(k), i)$.

Figure 5.1 visualises a simplified Balloon with a hash treap and a history tree. For the sake of readability, we omit the hash values and priority, replace hashed keys with integers, and replace hashed entries with place-holder labels. For example, the root in the hash treap has key 42 and value 1. The value 1 refers to the leaf node in the history tree with index 1, whose value is p42, the place-holder label for the hash of the entry which key, once hashed, is represented by integer 42.

By putting the hash of the entry key, $\texttt{Hash}(k)$, instead of the key into the hash treap, we avoid easy entry enumeration by third parties: no valid entry keys leak as part of authenticated paths in the treap for non-membership proofs. Note that when `H.MembershipGen` returns an entry, as specified in Section 5.1.2, the actual entry is retrieved from the data structure, not the hash of the entry as stored in the history tree (authentication). We store the hash of the entry in the history tree for sake of efficiency, since the entry is already stored in the (non-authenticated) data structure. Next, we describe the six algorithms that define Balloon as an ADS.

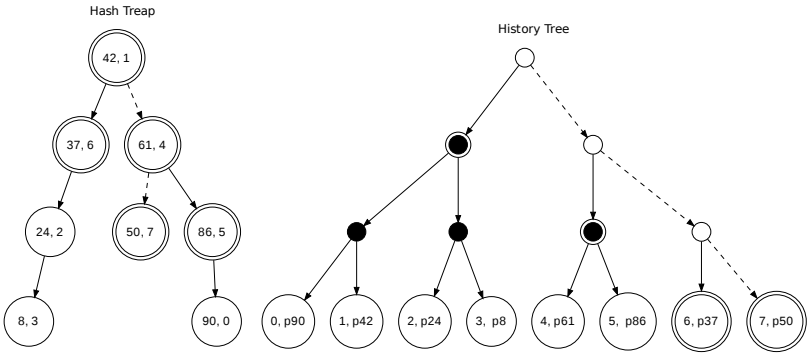


Figure 5.1 An simplified example of a Balloon consisting of a hash treap and history tree. A membership proof for an entry $e = (k, v)$ with $\texttt{Hash}(k) = 50$ and $\texttt{Hash}(e)$ denoted by p50 (place-holder label) consists of the circle nodes in both trees.

5.2.1 Setup

Algorithm $\{\texttt{sk}, \texttt{pk}\} \leftarrow \texttt{genkey}(1^{\lambda})$: Generates a signature key-pair $\{\texttt{sk}, \texttt{vk}\}$ using the generation algorithm of a signature scheme with security level λ and picks a function Ω that deterministically orders entries. Outputs the signing key as the private key $\texttt{sk} = \texttt{sk}$, and the verification key and the ordering function Ω as the public key $\texttt{pk} = \{\texttt{vk}, \Omega\}$.

Algorithm $\{\texttt{auth}(D_0), \texttt{s}_0\} \leftarrow \texttt{setup}(D_0, \texttt{sk}, \texttt{pk})$: Let D_0 be the initial data structure, containing the initial set of entries $< e_j >$. The authenticated data structure, $\texttt{auth}(D_0)$, is then computed by adding each entry from the set to the, yet empty, authenticated data structure in the order dictated by the function $\Omega \leftarrow \texttt{pk}$. The snapshot is defined as the root of the hash treap r and commitment in the history tree c_i for the entry that was added last together with a digital signature over those: $\texttt{s}_0 = \{r, c_i, \sigma\}$, where $\sigma = \texttt{Sign}_{\texttt{sk}}(\{r, c_i\})$.

5.2.2 Update and Refresh

Algorithm $\{D_{h+1}, \texttt{auth}(D_{h+1}), \texttt{s}_{h+1}, \texttt{upd}\} \leftarrow \texttt{update}(u, D_h, \texttt{auth}(D_h), \texttt{s}_h, \texttt{sk}, \texttt{pk})$: Let u be a set of entries to insert into D_h. The updated data structure D_{h+1} is the result of appending the entries in u to D_h and indicating that these belong to the $(h+1)^{\text{th}}$ set. The updated authenticated data structure, $\texttt{auth}(D_{h+1})$, is then computed by adding each entry from the set to the authenticated data structure $\texttt{auth}(D_h)$ in the order

dictated by the function $\Omega \leftarrow \texttt{pk}$. The updated snapshot is the root of the hash treap r and commitment in the history tree c_i for the entry that was added last together with a digital signature over those: $\texttt{s}_{h+1} = \{r, c_i, \sigma\}$, where $\sigma \leftarrow \texttt{Sign}_{\texttt{sk}}(\{r, c_i\})$. The update information contains this snapshot $\texttt{upd} = \texttt{s}_{h+1}$.

Algorithm $\{D_{h+1}, \texttt{auth}(D_{h+1}), \texttt{s}_{h+1}\} \leftarrow \texttt{refresh}(u, D_h, \texttt{auth}(D_h), \texttt{s}_h, \texttt{upd}, \texttt{pk})$: Let u be a set of entries to insert into D_h. The updated data structure D_{h+1} is the result of appending the entries in u to D_h and indicating that these belong to the $(h+1)^{\text{th}}$ set. The updated authenticated data structure, $\texttt{auth}(D_{h+1})$, is then computed by adding each entry from the set u to the authenticated data structure $\texttt{auth}(D_h)$ in the order dictated by the function $\Omega \leftarrow \texttt{pk}$. Finally, the new snapshot is set to $\texttt{s}_{h+1} = \texttt{upd}$.

5.2.3 Query and Verify

Algorithm $\{\Pi(q), \alpha(q)\} \leftarrow \texttt{query}(q, D_h, \texttt{auth}(D_h), \texttt{pk})$ (**Membership**): We consider the query q to be "a membership query for an entry with key k in the data structure that is fixed by $\texttt{s}_{\texttt{queried}}$", where $\texttt{queried} \leq h$. The query has two possible answers $\alpha(q)$: $\{\texttt{true}, e\}$ in case an entry e with key k exists in $D_{\texttt{queried}}$, otherwise `false`. The proof of correctness $\Pi(q)$ consists of up to three parts:

1. An authenticated path P^T in the hash treap to $k' = \texttt{Hash}(k)$.
2. The index i of the entry in the history tree.
3. A membership proof P on index i in the history tree.

The algorithm generates an authenticated path in the hash treap, which is part of $\texttt{auth}(D_h)$, to k': $\{P^T, v\} \leftarrow \texttt{T.AuthPath}(k')$. If $v \stackrel{?}{=} \texttt{null}$, then there is no entry with key k in D_h (and consequently in $D_{\texttt{queried}}$) and the algorithm outputs $\Pi(q) = P^T$ and $\alpha(q) = \texttt{false}$.

Otherwise, the value v in the hash treap indicates the index i in the history tree of the entry. Now the algorithm checks whether or not the index i is contained in the history tree up till $\texttt{auth}(D_{\texttt{queried}})$. If not, the algorithm outputs $\alpha(q) = \texttt{false}$ and $\Pi(q) = \{P^T, i\}$. If it is, the algorithm outputs $\alpha(q) = \{\texttt{true}, e_i\}$ and $\Pi(q) = \{P^T, i, P\}$, where $\{P, e_i\} \leftarrow \texttt{H.MembershipGen}(i, c_{\texttt{queried}})$ and $c_{\texttt{queried}} \leftarrow \texttt{s}_{\texttt{queried}}$.

Algorithm $\{\texttt{accept}, \texttt{reject}\} \leftarrow \texttt{verify}(q, \alpha, \Pi, \texttt{s}_h, \texttt{pk})$ (**Membership**): First, the algorithm extracts $\{k, \texttt{s}_{\texttt{queried}}\}$ from the query q and $\{P^T, i, P\}$ from Π, where i and P can be `null`. From the snapshot it extracts $r \leftarrow \texttt{s}_h$. Then the algorithm

computes $x \leftarrow P^T.\texttt{AuthPathVerify}(k,i)$. If $x \stackrel{?}{=} \texttt{false} \vee P^T.\texttt{root}() \neq r$, the algorithm outputs `reject`. The algorithm outputs `accept` if any of the following three conditions hold, otherwise `reject`:

- $\alpha \stackrel{?}{=} \texttt{false} \wedge i \stackrel{?}{=} \texttt{null}$.
- $\alpha \stackrel{?}{=} \texttt{false} \wedge i > \texttt{queried}[-1]$[4].
- $\alpha \stackrel{?}{=} \{\texttt{true}, e\} \wedge \texttt{key}(e) \stackrel{?}{=} k \wedge y \stackrel{?}{=} \texttt{true}$,
  for $y \leftarrow P.\texttt{MembershipVerify}(i, c_{\texttt{queried}}, \texttt{Hash}(e))$ and $c_{\texttt{queried}} \leftarrow \texttt{s}_{\texttt{queried}}$.

5.3 Correctness and Security

Theorem 6. *Balloon {*`genkey`,`setup`,`update`,`refresh`,`query`,`verify`*} is a correct ADS scheme for a data structure D, which contains a list of sets of entries, according to Definition 7, assuming the collision-resistance of the underlying hash function.*

Proof. For every membership `query` q and for every answer $\alpha(q)$ that is accepted by the `check` algorithm and proof output by `query`, `verify` accepts with overwhelming probability. Let q be a membership query for a key k in the snapshot $\texttt{s}_j$. Regardless of the queried snapshot, in the hash treap, the current root $r \in \texttt{s}_h$ is used to construct the authenticated path verified by `verify`.

If there is no entry with key k in D_h, then there is no key with $k' \leftarrow \texttt{Hash}(k)$ in the hash treap with overwhelming probability (a hash collision has negligible probability), and `verify` accepts only if $\alpha(q) \stackrel{?}{=} \texttt{false}$. If there is an entry with key k in D_h, then there is a node with key k' in the hash treap and $\alpha(q)$ is either $\{\texttt{true}, e\}$ or `false`. If $\alpha(q)$ is `false`, then the entry with key k was inserted into D_h *after* $\texttt{s}_j$. This means that the index i in the history tree, that is the value v' in the hash treap node for k', is $i > \texttt{j}[-1]$ and hence the `verify` algorithm accepts. Finally, if $\alpha(q)$ is $\{\texttt{true}, e\}$, then the proof contains a membership proof in the history tree for the key i and value $\texttt{Hash}(e)$ and hence the `verify` algorithm accepts. □

Theorem 7. *Balloon {*`genkey`,`setup`,`update`,`refresh`,`query`,`verify`*} is a secure ADS scheme for a data structure D, which contains a list of sets of entries, according to Definition 8, assuming the collision-resistance of the underlying hash function.*

[4] `queried`$[-1]$ denotes the index of the last inserted entry in version `queried` of the authenticated data structure.

Proof. The adversary initially outputs the authenticated data structure $\texttt{auth}(D_0)$ and the snapshot $\texttt{s}_0$ through an oracle call to algorithm `setup`. The adversary picks a polynomial number $i = 0, \ldots, h$ of updates with u_i insertions of unique entries and outputs the data structure D_i, the authenticated data structure $\texttt{auth}(D_i)$, and the snapshot $\texttt{s}_i$ through oracle access to `update`. Then it picks a query $q =$ "a membership query for an entry with key $k \in \{0,1\}^{|\texttt{Hash}(\cdot)|}$ in the data structure that is fixed by $\texttt{s}_j$, with $0 \leq j \leq h+1$", a proof $\Pi(q)$, and an answer $\alpha(q)$ which is rejected by $\texttt{check}(q, \alpha(q), D_j)$ as incorrect. An adversary breaks security if $\texttt{verify}(q, \alpha(q), \Pi(q), \texttt{s}_j, \texttt{pk})$ outputs `accept` with non-negligible probability.

Assume a probabilistic polynomial time adversary $\mathscr{A}$ that breaks security with non-negligible probability. Given that the different versions of the authenticated data structure and corresponding snapshots are generated through oracle access, these are correct, i.e., the authenticated data structure contains all elements of the data structure for each version, the root and commitment in each snapshot correspond to that version of the ADS and the signature in each snapshot verifies.

The tuple $(q, \alpha(q), D_j)$ is rejected by `check` in only three cases:

Case 1 $\alpha(q) = \texttt{false}$ and there exists an entry with key k in D_j.

Case 2 $\alpha(q) = \{\texttt{true}, e\}$ and there does not exists an entry with key k in D_j.

Case 3 $\alpha(q) = \{\texttt{true}, e\}$ and the entry e^* with key k in D_j differs from e: $e = (k, v) \neq e^* = (k, v^*)$ or more specifically $v \neq v^*$.

For all three cases where the `check` algorithm outputs `reject`, $\mathscr{A}$ has to forge an authenticated path in the hash treap and/or history tree in order to get the `verify` algorithm to output `accept`:

Case 1 In the hash treap that is fixed by $\texttt{s}_{h+1}$, there is a node with key $k' = \texttt{Hash}(k)$ and the value $v' \leq j[-1]$. However for the `verify` algorithm to output `accept` for $\alpha(q)$=`false`, the authenticated path in the hash treap must go to either no node with key k' or a node with key k' for which the value v' is greater than the index of the last inserted entry in the history tree that is fixed by $\texttt{s}_j$: $v' > j[-1]$.

Case 2 In the hash treap that is fixed by $\texttt{s}_{h+1}$, there is either no node with key $k' = \texttt{Hash}(k)$ or a node with key k' for which the value v' is greater than the index of the last inserted entry in the history tree that is fixed by $\texttt{s}_j$: $v' > j[-1]$. However for the `verify` algorithm to output `accept` for $\alpha(q) = \{\texttt{true}, e\}$, the

authenticated path in the hash treap must go to a node with key k', where the value $v' \leq j[-1]$. Note that, in this case, $\mathscr{A}$ also needs to forge an authenticated path in the history tree to succeed.

Case 3 In the hash treap that is fixed by $\mathtt{s}_{b+1}$, there is a leaf with key $k' = \mathtt{Hash}(k)$ and the value $v' \leq j[-1]$. In the history tree, the leaf with key v' has the value $\mathtt{Hash}(e^*)$. However for the `verify` algorithm to output `accept` for $\alpha(q) = \{\mathtt{true}, e\}$, the authenticated path in the hash treap must go to a leaf with key k', where the value $v' \leq j[-1]$, for which the authenticated path in the history tree must go to a leaf with key v' and the value $\mathtt{Hash}(e)$.

From Lemma 1 it follows that we can construct a probabilistic polynomial time adversary $\mathscr{B}$, by using $\mathscr{A}$, that outputs a collision of the underlying hash function with non-negligible probability. □

5.4 Verifiable Insert

In practical three-party settings, the source typically has less storage capabilities than servers. As such, it would be desirable that the source does not need to keep a copy of the entire (authenticated) data structure for `update`, but instead can rely on its own (constant) storage combined with verifiable information from a server. We define new query and verify algorithms that enable the construction of a *pruned* authenticated data structure, containing only the relevant structure for a set of entries to be inserted with a modified update algorithm. The pruned authenticated data structure is denoted by $\mathtt{pruned}\big(\mathtt{auth}(D_b), u\big)$, where $\mathtt{auth}(D_b)$ denotes the version of the ADS being pruned, and u the set of entries that this ADS is pruned for.

Algorithm $\{\Pi(q), \alpha(q)\} \leftarrow \mathtt{query}(q, D_b, \mathtt{auth}(D_b), \mathtt{pk})$ (**Prune**): We consider the query q to be "a prune query for if a set of entries u can be inserted into D_b". The query has two possible answers: $\alpha(q)$: `true` in case no key for the entries in u already exist in D_b, otherwise `false`. The proof of correctness $\Pi(q)$ either proves that there already is an entry with a key from an entry in u, or provides proofs that enable the construction of a pruned $\mathtt{auth}(D_b)$, depending on the answer. For every $k_j \leftarrow \mathtt{key}(e_j)$ in the set u, the algorithm uses as a sub-algorithm $\{\Pi'_j(q), \alpha'_j(q)\} \leftarrow \mathtt{query}(q'_j, D_b, \mathtt{auth}(D_b), \mathtt{pk})$ (**Membership**) with $q' = \{k_j, \mathtt{s}_b\}$, where $\mathtt{s}_b$ fixes $\mathtt{auth}(D_b)$. If any $\alpha'_j(q) \stackrel{?}{=} \mathtt{true}$, the algorithm outputs $\alpha(q) = \mathtt{false}$ and $\Pi(q) = \{\Pi'_j(q), k_j\}$ and stops. If not, the algorithm takes P_j^T from each $\Pi'_j(q)$ and creates the set $< P_j^T >$. Next, the algorithm extracts the latest

entry e_i inserted into the history tree from $\texttt{auth}(D_h)$ and uses as a sub-algorithm $\{\Pi'(q), \alpha'(q)\} \leftarrow \texttt{query}(q', D_h, \texttt{auth}(D_h), \texttt{pk})$ (**Membership**) with $q' = \{\texttt{key}(e_i), \texttt{s}_h\}$. The algorithm outputs $\alpha(q) = \texttt{true}$ and $\Pi(q) = \{< P_j^T >, \Pi'(q)\}$.

Algorithm $\{\texttt{accept}, \texttt{reject}\} \leftarrow \texttt{verify}(q, \alpha, \Pi, \texttt{s}_h, \texttt{pk})$ (**Prune**): The algorithm starts by extracting $< e_j > \leftarrow u$ from the query q. If $\alpha \stackrel{?}{=} \texttt{false}$, it gets $\{\Pi'_j(q), k_j\}$ from Π and uses as a sub-algorithm $\texttt{valid} \leftarrow \texttt{verify}(q', \alpha', \Pi', \texttt{s}_h, \texttt{pk})$ (**Membership**), with $q' = \{k, \texttt{s}_h\}$, $\alpha' = \texttt{true}$ and $\Pi' = \Pi'_j(q)$, where $k \leftarrow k_j$. If $\texttt{valid} \stackrel{?}{=} \texttt{accept}$ and there exists an entry with key k in u, the algorithm outputs `accept`, otherwise `reject`.

If $\alpha \stackrel{?}{=} \texttt{true}$, extract $\{< P_j^T >, \Pi'(q)\}$ from Π. For each entry e_j in u, the algorithm gets $k_j \leftarrow \texttt{key}(e_j)$, finds the corresponding P_j^T for $k'_j = \texttt{Hash}(k_j)$, and uses as a sub-algorithm $\texttt{valid} \leftarrow \texttt{verify}(q', \alpha', \Pi', \texttt{s}_h, \texttt{pk})$ (**Membership**), with $q' = \{k_j, \texttt{s}_h\}$, $\alpha' = \texttt{false}$ and $\Pi' = P_j^T$. If no corresponding P_j^T to k'_j is found in $< P_j^T >$ or $\texttt{valid} \stackrel{?}{=} \texttt{reject}$, then the algorithm outputs `reject` and stops. Next, the algorithm uses as a sub-algorithm $\texttt{valid} \leftarrow \texttt{verify}(q', \alpha, \Pi', \texttt{s}_h, \texttt{pk})$ (**Membership**), with $q' = \{\texttt{key}(e_i), \texttt{s}_h\}$ and $\Pi' = \Pi'(q)$, where $e_i \in \Pi'(q)$. If $\texttt{valid} \stackrel{?}{=} \texttt{accept}$ and $i \stackrel{?}{=} h[-1]$ the algorithm outputs `accept`, otherwise `reject`.

Algorithm $\{\texttt{s}_{h+1}, \texttt{upd}\} \leftarrow \texttt{update*}(u, \Pi, \texttt{s}_h, \texttt{sk}, \texttt{pk})$: Let u be a set of entries to insert into D_h and Π a proof that the sub-algorithm $\texttt{verify}(q, \alpha, \Pi, \texttt{s}_h, \texttt{pk})$ (**Prune**) outputs `accept` for, where $q = u$ and $\alpha = \texttt{true}$. The algorithm extracts $\{< P_j^T >, \Pi'(q)\}$ from Π and builds a pruned hash treap $\texttt{pruned(T)} \leftarrow \texttt{BuildPrunedTree}(< P_j^T >)$. Next, it extracts P from $\Pi'(q)$ and constructs the pruned Balloon $\texttt{pruned}\big(\texttt{auth}(D_h), u\big) \leftarrow \{\texttt{pruned(T)}, P\}$. Finally, the algorithm adds each entry in u to the pruned Balloon $\texttt{pruned}\big(\texttt{auth}(D_h), u\big)$ in the order dictated by $\Omega \leftarrow \texttt{pk}$. The updated snapshot is the digital signature over the root of the updated pruned hash treap r and commitment in the updated pruned history tree c_i for the entry that was added last: $\texttt{s}_{h+1} = \{r, c_i, \sigma\}$, where $\sigma \leftarrow \texttt{Sign}_{\texttt{sk}}(\{r, c_i\})$. The update information contains this snapshot $\texttt{upd} = \texttt{s}_{h+1}$.

Lemma 2. *The output of* `update` *and* `update*` *is identical with respect to the root of the hash treap and the latests commitment in the history tree of* $\texttt{s}_{h+1}$ *and* upd[5].

Proof. For `verify` (**Prune**), when $\alpha \stackrel{?}{=} \texttt{true}$, the algorithm verifies $|u|+1$ membership queries using as a sub-algorithm `verify` (**Membership**), which is correct and secure according to Theorems 6 and 7. Furthermore, `verify` (**Prune**) verifies that no entry

[5]Note that the signatures may differ since the signature scheme can be probabilistic.

in u has already been inserted into the Balloon and that a membership query fixes the last entry inserted into the history tree. This enables `update*` to verifiably create $\texttt{pruned}\big(\texttt{auth}(D_b), u\big) \leftarrow \{\texttt{pruned}(\mathrm{T}), P\}$. Next we show that inserting entries u into $\texttt{pruned}\big(\texttt{auth}(D_b), u\big)$ and $\texttt{auth}(D_b)$ result in the same root r of the hash treap and commitment c_i on the history tree.

The pruned hash treap `pruned(T)` is the only part of the hash treap that is subject to change when inserting the entries u. This follows from the fact that the position to insert each a new node in a treap is determined by a binary search (which path a membership query fixes and proves), and that balancing the treap using the heap priority only rotates nodes along the authenticated path. Since a treap is also set unique, i.e., independent of the insert order of entries, all these authenticated paths can safely be combined into one pruned tree. As such the root of the hash treap after inserting the entries will be identical when using the pruned and full hash treap.

To compute the new root of a Merkle tree after adding a leaf node, one needs the authenticated path of the last inserted leaf node to the root and the new leaf. Note that any new leaf will always be inserted to the right of the authenticated path. As such the roots (and by consequence the commitments) of the new history tree after adding all entries in the order determined by $\Omega \leftarrow \texttt{pk}$ will be identical for both cases. □

As a result of Lemma 2, the `update` algorithm in Balloon can be replaced by `update*` without breaking the correctness and security of the Balloon as in Theorems 6 and 7. This means that the server can keep and refresh the (authenticated) data structure while the controller only needs to store the last snapshot s_b to be able to produce updates, resulting in a small constant size storage requirement for the controller.

Note that, in order to save on transmission bandwidth, `verify` (**Prune**) could output the pruned authenticated data structure directly. Given that $\texttt{pruned(T)}.\texttt{AuthPathVerify}(k, v)$ and $P^T.\texttt{AuthPathVerify}(k, v)$ are equivalent, the correctness and security of `verify` (**Prune**) reduce to `verify`. Section 5.6 shows how much bandwidth can be saved.

5.5 Publicly Verifiable Consistency

While the server is untrusted, the controller is trusted. A stronger adversarial model assumes forward security for the controller: the controller is only trusted up to a certain point in time, i.e., the time of compromise, and afterwards cannot change

the past. In this stronger adversarial model, Balloon should still provide correctness and security for all entries inserted by the controller up till the time of controller compromise.

Efficient incremental proofs, realised by the `IncGen` and `IncVerify` algorithms, are a key feature of history trees [37]. Anyone can challenge the server to provide a proof that one commitment as part of a snapshot is *consistent* with all previous commitments as part of snapshots. However, it appears to be an open problem to have an efficient algorithm for showing consistency between roots of different versions of a treap (or any lexicographically sorted data structure) [40]. In Section 5.8, we show why one cannot efficiently use probabilistic proofs of consistency for a Balloon. In absence of efficient (both for the server and verifier in terms of computation and size) incremental proofs in hash treaps, we rely on a concept from Certificate Transparency [72]: monitors.

We assume that a subset of clients, or any third party, will take on a role referred to as a "monitor", "auditor", or "validator" in ,e.g., [15, 66, 71, 72, 101, 125]. A monitor continuously monitors all data stored at a server and ensures that all snapshots issued by a controller are consistent. We assume that clients and monitors receive the snapshots through gossiping.

Definition 9 (Publicly Verifiable Consistency)**.** *An ADS scheme is publicly verifiable consistent if anyone can verify that a set of entries u has been correctly inserted in D_h and* $\texttt{auth}(D_h)$*, fixed by* $\texttt{s}_h$ *to form* D_{h+1} *and* $\texttt{auth}(D_{h+1})$ *fixed by* $\texttt{s}_{h+1}$*.*

Algorithm $\{\alpha, D_{h+1}, \texttt{auth}(D_{h+1}), \texttt{s}_{h+1}\} \leftarrow \texttt{refreshVerify}(u, D_h, \texttt{auth}(D_h), \texttt{s}_h, \texttt{upd}, \texttt{pk})$: First, the algorithm runs $\{D_{h+1}, \texttt{auth}(D_{h+1}), \texttt{s}_{h+1}\} \leftarrow \texttt{refresh}(u, D_h, \texttt{auth}(D_h), \texttt{s}_h, \texttt{upd}, \texttt{pk})$ as a sub-algorithm. Then, the algorithm verifies the updated snapshot $\{r, c_i, \sigma\} \leftarrow \texttt{s}_{h+1} \leftarrow \texttt{upd}$:

1. Verify the signature: $\texttt{true} \stackrel{?}{=} \texttt{Verify}_{\texttt{pk}}(\sigma, \{r, c_i\})$.
2. Match the root of the updated hash treap $r' \stackrel{?}{=} r$.
3. Match the last commitment in the updated history tree $c_i' \stackrel{?}{=} c_i$.

If the verification succeeds, the algorithm outputs $\{\alpha = \texttt{true}, D_{h+1}, \texttt{auth}(D_{h+1}), \texttt{s}_{h+1}\}$. Otherwise, the algorithm outputs $\alpha = \texttt{false}$.

Theorem 8. *With* `refreshVerify`*, Balloon is publicly verifiable consistent according to Definition 9, assuming perfect gossiping of the snapshots and the collision-resistance of the underlying hash function.*

Proof. By assuming perfect gossiping, one is assured that it receives all snapshots in the order they were generated and that these snapshots have not been altered afterwards.

First, one starts from the initial data structure D_0 and constructs the initial authenticated data structure $\texttt{auth}(D_0)$ by adding the entries contained in D_0 to an empty Balloon. It then verifies the initial snapshot $\texttt{s}_0$ (received by gossip) as in `refreshVerify`. If snapshot verifies, one now has $\{D_0, \texttt{auth}(D_0), \texttt{s}_0\}$ and is assured these values are consistent with the (authenticated) data structure as constructed by the controller. Under the assumption that the underlying hash function is collision resistant, this follows directly from Lemma 1 and the fact that our authenticated data structure consists of two Merkle trees: a hash treap and a history tree.

Now, one keeps on building the authenticated data structure, snapshot by snapshot, until one finally ends up with the snapshots that one wants to check the consistency between. The authenticated data structure is built by running $\texttt{refreshVerify}(u, D_i, \texttt{auth}(D_i), \texttt{s}_i, \texttt{s}_{i+1}, \texttt{pk})$, where every time i is increased by one. A balloon is verifiably consistent, if for $i = h$, this algorithm outputs $\{D_{h+1}, \texttt{auth}(D_{h+1}), \texttt{s}_{h+1}\}$. If the balloon is not consistent, then the output of the `refreshVerify` algorithm is $\alpha = \texttt{false}$ for some $i \leq h$. □

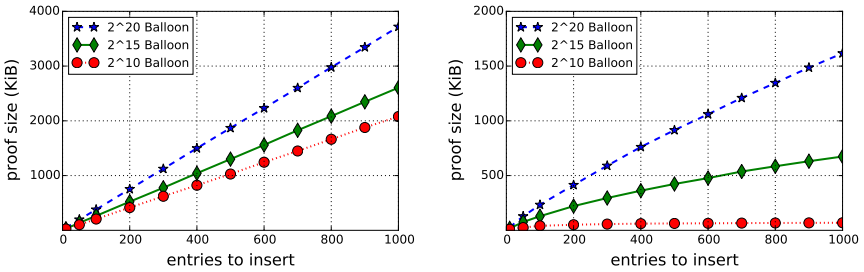
Note that for the purpose of verifying consistency between snapshots, it is not necessary to keep the data structure D. Moreover, the storage requirement for monitors can be further reduced by making use of pruned versions of the authenticated data structure, i.e., by using a $\texttt{refresh}^*$ sub-algorithm, similar to the $\texttt{update}^*$ algorithm. Finally, to preserve entry privacy towards monitors, one can provide the monitors with $\tilde{u} =< \tilde{e}_j >$, where $\tilde{e}_j = \big(\texttt{Hash}(k_j), \texttt{Hash}(e_j)\big)$, and not the actual set of entries. However, in this case, one must ensure that the ordering function $\Omega \leftarrow \texttt{pk}$ provides the same output for u and $\tilde{u}$.

5.6 Performance

We implemented Balloon in the Go[6] programming language using SHA-512 as the hash function (to match the use of NaCl in the following chapter). The output of SHA-512 is truncated to 256-bits, with the goal of reaching a 128-bit security level. Our performance evaluation focuses on verifiable inserts, that is composed of performing and verifying $|u| + 1$ membership queries, since these algorithms presumably are the most common. Figure 5.2 shows the size of the proof from `query` (**Prune**) in

[6] https://golang.org, accessed 2015-04-10.

KiB based on the number of entries to insert ranging from 1–1000 for three different sizes of Balloon: 2^{10}, 2^{15}, and 2^{20} entries. Figure 5.2a includes redundant nodes in the membership query proofs, and shows that the proof size is linear with the number of entries to insert. Figure 5.2b excludes redundant nodes between proofs, showing that excluding redundant nodes roughly halves the proof size with bigger gains the more entries are inserted. For large Balloons the probability that any two authenticated paths in the hash treap share nodes goes down, resulting in bigger proofs, until the number of entries get closer to the total size of the Balloon, when eventually all nodes in the hash treap are included in the proof as for the 2^{10} Balloon.



(a) Including redundant nodes. **(b)** Excluding redundant nodes.

Figure 5.2 The size of the proof from `query` (**Prune**) in KiB based on the number of entries to insert $|u|$ for different sizes of Balloon.

Table 5.1 shows a micro-benchmark of the three algorithms that enable verifiable inserts: `query`(**Prune**), `verify`(**Prune**), and `update*`. The table shows the average insert time (ms) calculated by Go's built-in benchmarking tool that performed between 30–30000 samples per measurement. The `update*` algorithm performs the bulk of the work, with little difference between the different Balloon sizes, and linear scaling for all three algorithms based on the number of entries to insert.

5.7 Related Work

Balloon is closely related to authenticated dictionaries [84] and persistent authenticated dictionaries (PADs) [8, 39, 40]. Balloon is not a PAD because it does not allow for the controller to remove or update keys from the data structure, i.e., it is append-only. By allowing the removal of keys, the server needs to be able to construct past versions of the PAD to calculate proofs, which is relatively costly. In Table 5.2,

Table 5.1 A micro-benchmark on Debian 7.8 (x64) using an Intel i5-3320M quad core 2.6GHz CPU and 7.7 GB DDR3 RAM.

Average time (ms)	Balloon 2^{10}			Balloon 2^{15}			Balloon 2^{20}		
	# Entries $\lvert u \rvert$			# Entries $\lvert u \rvert$			# Entries $\lvert u \rvert$		
	10	100	1000	10	100	1000	10	100	1000
`query` (**Prune**)	0.04	0.37	3.64	0.04	0.37	3.64	0.06	0.37	3.62
`verify` (**Prune**)	0.07	0.72	6.83	0.07	0.73	6.84	0.07	0.72	6.85
`update*`	0.56	4.45	41.2	0.78	5.76	50.3	0.90	6.35	52.7

Balloon is compared to the most efficient tree-based PAD construction according to Crosby & Wallach [40]: a red-black tree using Sarnak-Tarjan versioned nodes with a cache-everywhere strategy for calculated hash values. The table shows expected complexity. Note that red-black trees are more efficient than treaps due to their worst-case instead of expected logarithmic bounds on several important operations. We opted for using a treap due to its relative simplicity. For Balloon, the storage at the controller is constant due to using verifiable inserts, while the PAD maintains a copy of the entire data structure. To query past versions, the PAD has to construct past versions of the data structure, while Balloon does not. When inserting new entries, the PAD has to store a copy of the modified authenticated path in the red-black tree, while the storage for Balloon is constant. However, Balloon is less efficient when inserting new entries with regard to the proof size due to verifiable inserts.

Table 5.2 Comparing Balloon and an efficient PAD construction [40]. The number of entries in the data structure is n and the size of the version cache is v.

Expected Complexity	Total Storage Size (C)	Query Time (current)	Query Time (past)	Insert Storage Size (S)	Insert Time (C)	Insert Time (S)	Insert Proof Size
Balloon	$\mathcal{O}(1)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$
Tree-based PAD	$\mathcal{O}(n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log v \cdot \log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$

Miller et al. [81] present a generic method for authenticating operations on any data structure that can be defined by standard type constructors. The prover provides the authenticated path in the data structure that is traversed by the prover when performing an operation. The verifier can then perform the same operation, only needing the authenticated paths provided in the proof. The verifier only has to store the latest correct digest that fixes the content of the data structure. Our verifiable insert is based on the same principle.

Secure logging schemes, like the work by Schneier and Kelsey [106], Ma and Tsudik [74], and Yavuz *et al.* [124] can provide deletion-detection forward integrity in a forward secure adversary model for append-only data. Some schemes, like that of Yavuz *et al.*, are publicly verifiable like Balloon. However, these schemes are insufficient in our setting, since clients cannot get efficient non-membership proofs or efficient membership-proofs for past versions of the data structure when only some versions (snapshots) are timestamped.

All the following related work operates in a setting that is fundamentally different to the one of Balloon. For Balloon, we assume a forward-secure controller with an untrusted server, whereas the following related work assumes a (minimally) trusted server with untrusted controllers.

Certificate Transparency (CT) by Laurie *et al.* [72] and the tamper-evident history system by Crosby & Wallach [38] use a nearly identical[7] data structure and operations. Even though in both Certificate Transparency and Crosby & Wallach's history system, a number of *minimally trusted* controllers insert data into a history tree kept by a server, clients query the server for data and can act as auditors or monitors to challenge the server to prove consistency between commitments. Non-membership proofs require the entire data structure to be sent to the verifier.

In Revocation Transparency, Laurie and Kasper [71] present the use of a sparse Merkle tree for certificate revocation. Sparse Merkle trees create a Merkle tree with 2^N leafs, where N is the bit output length of a hash algorithm. A leaf is set to 1 if the certificate with the hash value fixed by the path to the leaf from the root of the tree is revoked, and `0` if not. While the tree in general is too big to store or compute on its own, the observation that most leafs are zero (i.e., the tree is sparse), means that only paths including non-zero leafs need to be computed and/or stored. At first glance, sparse Merkle trees could replace the hash treap in a Balloon with similar size/time complexity operations.

Enhanced Certificate Transparency (ECT) by Ryan [101] extends CT by using two data structures: one chronologically sorted and one lexicographically sorted. Distributed Transparent Key Infrastructure (DTKI) [125] builds upon the same data structures as ECT. The chronologically sorted data structure corresponds to a history tree (like CT). The lexicographically sorted data structure is similar to our hash treap. For checking consistency between the two data structures, ECT and DTKI use probabilistic checks. The probabilistic checking verifies that a random operation recorded in the chronological data structure has been correctly performed

[7] The difference is in how non-full trees are handled, as noted in Section 2.1 of [72].

in the lexicographical data structure. However, this requires the prover to generate past versions of the lexicographical data structure (or cache all proofs), with similar trade-offs as for PADs, which is relatively costly.

CONIKS [78] is a privacy-friendly key management system where minimally trusted clients manage their public keys in directories at untrusted key servers. A directory is built using an authenticated binary prefix tree, offering similar properties as our hash treap. In CONIKS, user identities are presumably easy to brute-force, so they go further than Balloon in providing entry privacy in proofs by using verifiable unpredictable functions and commitments to hide keys (identities) and values (user data). CONIKS stores every version of their (authenticated) data structure, introducing significant overhead compared to Balloon. However, CONIKS supports modifying and removing keys, similar to a PAD. Towards consistency, CONIKS additionally links snapshots together into a snapshot chain, together with a specified gossiping mechanism that greatly increases the probability that an attacker creating inconsistent snapshots is caught. This reduces the reliance on perfect gossiping, and could be used in Balloon. If the controller ever wants to create a fork of snapshots for a subset of clients and monitors, it needs to maintain this fork forever for this subset or risk detection. Like CONIKS, we do not prevent an adversary compromising a server, or controller, or both, from performing attacks: we provide means of detection after the fact.

5.8 Negative Result on Probabilistic Consistency

Here we present a negative result from our attempt at ensuring consistency of a Balloon with probabilistic proofs. Probabilistic proofs are compelling, because they may enable more resource-constrained clients en-mass to verify consistency, removing the need for monitors that perform the relatively expensive role of downloading all entries at a server. Assume the following pair of algorithms:

- $P \leftarrow \texttt{B.IncGen}(\texttt{s}_i, \texttt{s}_j, \texttt{rand})$: Generates a probabilistic incremental proof P using randomness `rand` between $\texttt{s}_i$ and $\texttt{s}_j$, where $i \leq j$, from the Balloon B. Outputs P.

- $\{\texttt{accept}, \texttt{reject}\} \leftarrow \texttt{P.IncVerify}(s_i, \texttt{s}_j, \texttt{rand})$: Verifies that P probabilistically proves that $\texttt{s}_j$ fixes every entry fixed by s_i, where $i \leq j$, using randomness `rand`.

5.8.1 Our Attempt

Our envisioned `B.IncGen` algorithm shows consistency in two steps. First, it uses the `H.IncGen` algorithm from the history tree. This ensures that the snapshots are consistent for the history tree. Second, it selects deterministically and uniformly at random based on `rand`, a number of entries $E =< e_j >$ from the history tree. Which entries to select from depend on the two snapshots. For each selected entry, the algorithm performs a `query` (**Membership**) for the entry key $k_j \leftarrow \texttt{key}(e_j)$ to show that the entry is part of the hash treap and points to the index of the entry in the history tree.

The `P.IncVerify` algorithm checks the incremental proof in the history tree, `verify` (**Membership**) $\stackrel{?}{=}$ `accept` for each output of `query` (**Membership**), and that the entries E were selected correctly based on `rand`. Next, we explain an attack, why it works, and possible lessons learned.

5.8.2 Attack

Here we explain an attack on our attempt that allows an attacker to hide an arbitrary entry that was inserted *before controller compromise*. The attacker takes control over both the controller and server just after snapshot s_t. Assume that the attacker wants to remove an entry e_j from Balloon, where $j \leq t[-1]$. The attacker does the following:

1. Remove the entry key $k'_j = \texttt{Hash}(k_j)$, where $k_j \leftarrow \texttt{key}(e_j)$ from the hash treap and insert a random key and rebalance the treap if needed. This results is a modified ADS $\texttt{auth}(D_t)^*$.

2. Generate a set of new entries u and update the data structure: $\texttt{update}(u, D_t,$ $\texttt{auth}(D_t)^*, \texttt{s}_t, \texttt{sk}, \texttt{pk})$, resulting in a new snapshot $\texttt{s}_{t+1}$.

It is clear that the snapshot $\texttt{s}_{t+1}$ is inconsistent with all other prior snapshots, $\texttt{s}_p$, where $p \leq t$.

Now, we show how the attacker can avoid being detected by `P.IncVerify` in the case that the verifier challenges the server (and therefore the attacker) to probabilistically prove the consistency between $\texttt{s}_p$ and $\texttt{s}_{t+1}$, AND that the randomness `rand` provided by the verifies selects the entry e_j that was modified by the attacker. The attacker can provide a valid incremental proof in the history tree, using `H.IncGen`, since the history tree has not been modified. However, the attacker cannot create a valid membership proof for an entry with key $k_j \leftarrow \texttt{key}(e_j)$ in the ADS, since the key $k'_j = \texttt{Hash}(k_j)$ was removed from the hash treap in $\texttt{auth}(D_{t+1})$. To avoid

detection, the attacker puts back the entry key k'_j in the hash treap and rebalances the treap if needed. Now by inserting a set of entries using `update`, a *new snapshot* s_{t+2} is generated, which is then used to perform the membership query against that will now output a valid membership proof.

5.8.3 Lessons Learnt

This attack succeeds because the attacker can, once having compromised the controller and server, a) create snapshots at will; and b) membership queries are always performed on the current version of the hash treap.

In settings where snapshots are generated periodically, e.g., once a day, the probability of the attacker getting caught in this way is non-negligible given a sufficient number of queries. However, as long as the attacker can create snapshots at will, the probability that it will be detected with probabilistic incremental proofs is zero, as long as it cannot be challenged to generate past versions of the hash treap; and there are no monitors or another mechanism, that prevent the attacker from modifying or deleting entries that were inserted into the ADS prior to compromise.

5.9 Conclusions

This chapter presented Balloon, an authenticated data structure composed of a history tree and a hash treap, which is tailored for privacy-preserving transparency logging. Balloon is a provably secure authenticated data structure, using a similar approach as Papamanthou *et al.* [86], under the modest assumption of a collision-resistant hash function. Balloon also supports efficiently verifiable inserts of new entries and publicly verifiable consistency. Verifiable inserts enable the controller to discard its copy of the (authenticated) data structure, only keeping constant storage, at the cost of transmitting and verifying proofs of a pruned version of the authenticated data structure. Publicly verifiable consistency enables anyone to verify the consistency of snapshots, laying the foundation for a forward-secure controller, under the additional assumption of a perfect gossiping mechanism of snapshots. Balloon is practical, as shown in Section 5.6, and a more efficient solution in our setting than using a PAD, as summarised by Table 5.2. Next, we use Balloon to construct a privacy-preserving one-way messaging scheme that can be used for transparency logging.

Chapter 6

Insynd: Privacy-Preserving One-Way Messaging Using Balloons

> Cyberspace. A consensual hallucination experienced daily by billions of legitimate operators, in every nation, by children being taught mathematical concepts... A graphic representation of data abstracted from the banks of every computer in the human system. Unthinkable complexity. Lines of light ranged in the nonspace of the mind, clusters and constellations of data. Like city lights, receding.
>
> *William Gibson* – Neuromancer

In this chapter we generalise transparency logging to *one-way messaging*. This is motivated by relating the setting to more areas in the literature than secure logging, potentially opening up for future work to benefit from work done in these settings.

Insynd[1] is a cryptographic scheme for one-way messaging that is designed to be secure and privacy-preserving. The development of Insynd has been motivated by the need for protecting one-way communication by controllers to potentially offline users where technologies like e-mail, SMS, and push notifications fall short due to lack of adequate privacy and security protections. For direct communication with controllers, users already have access to, e.g., TLS, Tor [45], and Tor hidden services that provide different security and privacy protection for both users and services. These technologies assume two online communicating parties, and therefore are not directly a fit for asynchronous communication. Ongoing work, e.g., Pond by Langley[2], builds asynchronous communication on top of technologies like Tor and Tor hidden services, with the goal of addressing several pressing security and privacy issues in *two-way* asynchronous communication where both the sender and the receiver may be unavailable, similar to email. While closely related to our setting, we are addressing the need of *one-way* asynchronous communication from a controller with high availability to less reliable users.

If controllers are available and users are not, the natural question is then why there is a need to communicate asynchronously at all? The user can just contact the controller, using technologies like Tor and TLS for privacy and security protection, at their convenience. However, the controller might get compromised between the time of generating messages and the time of users retrieving those messages. Therefore, Insynd supports safely storing messages on intermediate servers, such that controllers can generate messages independent of users. Since these servers do not need to be trusted, a controller can either run the server on its own or safely outsource it to, e.g., commodity cloud services. This means that we look at *how* the controller should store messages. Insynd uses an authenticated data structure, namely Balloon from Chapter 5, to safely outsource storage.

The rest of the chapter is structured as follows. Section 6.1 provides an overview of our setting, adversary model, assumptions, and goals. Section 6.2 introduces the cryptographic building blocks used in Insynd. Section 6.3 presents the cryptographic scheme Insynd, consisting of five protocols that are inspired by concepts from authenticated data structures, secure logging, and ongoing work on secure messaging protocols. Section 6.4 evaluates the properties of Insynd. Section 6.5 presents related work. We show that our proof-of-concept implementation offers comparable perfor-

[1] Insynd is a word-play on the Swedish word "insyn", roughly translatable to "insight", with a "d" suffix in the tradition of a deamon. The result, "Insynd", can be seen in Swenglish as "in sin", since "synd" means "sin" in Swedish.

[2] https://pond.imperialviolet.org/, accessed 2015-01-29.

mance for entry generation to state-of-the-art secure logging systems in Section 6.6. Section 6.7 concludes this chapter.

6.1 Overview

We take an entry-centric view, where an entry is a container for a message (the exact format of an entry is specified in Section 6.3.2). An entry is generated by a controller C, and intended for a user U. Entries are sent by the controller C to an intermediate server S, and the user U polls the server for new entries. We consider a *forward secure* [19] controller, where our goal is to protect entries sent *prior to compromise* of the controller. Furthermore, we distinguish between two types of compromises: a compromise by an active adversary and a passive adversary, where the compromise of the passive adversary is limited in time. If the compromise is by an active adversary, we provide forward security. If the compromise is time-limited by a passive adversary, we can recover once the controller is no longer compromised. The server is considered untrusted, and therefore for the purposes of our evaluation later considered controlled by an active adversary. This change requires us to also change our notion of privacy from the previous chapters, in particular when it comes to unlinkability of entries. We now focus on unlinkability of entries for *each run* of a protocol to insert new entries, since the adversary will be able to distinguish between each run of the protocol.

6.1.1 Threats and Assumptions

The ability to recover from a time-limited compromise of a passive adversary protects against a large number of threats. For example, a memory dump of the controller's system, which could happen for forensic analysis, as a result of a system crash, a privileged attacker looking for secrets in memory, lost or compromised backups, and legal obligations to provide data to law enforcement or other legal entities. Protocols like Off-the-Record Messaging (OTR) [30] provide protections against similar threats.

Commodity cloud services, while convenient, pose a large threat to both security and privacy due to the inherent loss of control over data and processing. As mentioned before, treating the server as untrusted covers both the case when the controller wants to store its entries itself and when the controller wants to outsource storage.

For communication, we assume a secure channel between the controller and the server (such as TLS), and a secure and anonymous channel for users (such as TLS

over Tor) to communicate with the controller and server. We explicitly consider availability out of scope, that is, the controller and server will always reply (however, their replies may be malicious). For time-stamps, we assume there exists a trustworthy time-stamping authority [32].

6.1.2 Properties and Goals

We note the following properties and goals for Insynd:

Secrecy Only the intended user of a message can read it.

Deletion-detection forward integrity Nobody can modify or delete messages sent prior to controller compromise without detection.

Forward unlinkability of entries For each run by the controller of the protocol to send new messages, all the entries sent in that run are unlinkable. This also implies that an adversary cannot tell which entries belong to which user, thus preventing user profiling due to entry generation.

Publicly verifiable consistency Anyone should be able to verify the consistency of all entries stored at a server.

Publicly verifiable proofs Both the controller and user receiving a message can create publicly verifiable proofs. The proofs are publicly verifiable in the sense that anyone can verify the proof with publicly available information (like a verification key). Insynd can generate the following publicly verifiable proofs:

Controller Who was the controller that created the entry.

Time When was the entry sent.

User Who was the recipient user of the entry.

Message What is the message in the entry.

These proofs should not inadvertently require the disclosure of private keys. Each proof is an isolated disclosure and potential violation of a property of Insynd, like message secrecy.

Non-interactive registration A controller can enable another controller to send messages to users already registered with the initiating controller. This enables distributed settings with multiple controllers, where controllers do not need to

interact with users to start sending messages. Furthermore, the identifiers for the user at the two controllers are unlinkable for sake of privacy.

Ease of implementation Primitives should be chosen in order to ease the implementation as much as possible, not shift unnecessary complexity from the designers to the implementers.

6.2 Building Blocks

The general idea behind Insynd is to store entries at an untrusted server with the help of Balloon (an authenticated data structure [77]) from Chapter 5. Each entry consists of a unique identifier that is only reconstructible by the intended user, and an encrypted message for that user. Balloon enables us to support a stronger adversary model than earlier work presented in Chapter 3, and it is crucial in providing our publicly verifiable proofs. Next, we summarise Balloon, describe a forward secure state generation mechanism, and the requirements we have on the public key encryption scheme to be able to reach the goals set in Section 6.1.2. Finally, we present and motivate our selection of the selected cryptographic primitives.

6.2.1 Balloon

We use Balloon from Chapter 5 as a key building block. We use the following algorithms for Insynd with a "*B*." prefix to distinguish the algorithms on Balloon from others later defined for Insynd:

- $\{\texttt{sk},\texttt{pk}\} \leftarrow \texttt{B.genkey}(1^\lambda)$: On input the security parameter λ, it outputs a secret key sk and public key pk.
- $\{\texttt{auth}(D_0),\texttt{s}_0\} \leftarrow \texttt{B.setup}(D_0,\texttt{sk},\texttt{pk})$: On input a (plain) data structure D_0, the secret key sk, and the public key pk, it computes the authenticated data structure $\texttt{auth}(D_0)$ and the corresponding snapshot $\texttt{s}_0$.
- $\{D_{b+1},\texttt{auth}(D_{b+1}),\texttt{s}_{b+1}\} \leftarrow \texttt{B.refresh}(u,D_b,\texttt{auth}(D_b),\texttt{s}_b,\texttt{upd},\texttt{pk})$: On input an update u on the data structure D_b, the authenticated data structure $\texttt{auth}(D_b)$, the snapshot $\texttt{s}_b$, relative information upd and the public key pk, it outputs the updated data structure D_{b+1} along with the updated authenticated data structure $\texttt{auth}(D_{b+1})$ and the updated snapshot $\texttt{s}_{b+1}$.

- $\{\Pi(q), \alpha(q)\} \leftarrow \texttt{B.query}(q, D_b, \texttt{auth}(D_b), \texttt{pk})$ (**Membership**): On input a membership query q on data structure D_b, the authenticated data structure $\texttt{auth}(D_b)$ and the public key pk, it returns the answer $\alpha(q)$ to the query, along with proof $\Pi(q)$.

- $\{\texttt{accept}, \texttt{reject}\} \leftarrow \texttt{B.verify}(q, \alpha, \Pi, \texttt{s}_b, \texttt{pk})$ (**Membership**): On input membership query q, an answer α, a proof Π, a snapshot $\texttt{s}_b$ and the public key pk, it outputs either accept or reject.

- $\{\Pi(q), \alpha(q)\} \leftarrow \texttt{B.query}(q, D_b, \texttt{auth}(D_b), \texttt{pk})$ (**Prune**) On input a prune query q on data structure D_b, the authenticated data structure $\texttt{auth}(D_b)$ and the public key pk, it returns the answer $\alpha(q)$ to the query, along with proof $\Pi(q)$.

- $\{\texttt{accept}, \texttt{reject}\} \leftarrow \texttt{B.verify}(q, \alpha, \Pi, \texttt{s}_b, \texttt{pk})$ (**Prune**): On input prune query q, an answer α, a proof Π, a snapshot $\texttt{s}_b$ and the public key pk, it outputs either accept or reject.

- $\{\texttt{s}_{b+1}, \texttt{upd}\} \leftarrow \texttt{B.update*}(u, \Pi, \texttt{s}_b, \texttt{sk}, \texttt{pk})$: On input an update u for the (authenticated) data structure fixed by $\texttt{s}_b$ with an accepted verified prune proof Π, the secret key sk, and the public key pk, it outputs the updated snapshot $\texttt{s}_{b+1}$, that fixes an update of the data structure D_{b+1} along with the updated authenticated data structure $\texttt{auth}(D_{b+1})$ using u, and some relative information upd.

We note that at its core Balloon depends on a pre-image and collision resistant hash function, and an unforgeable signature algorithm as long as the controller (source) remains uncompromised. We assume the existence of monitors and some form of gossiping mechanism. We do not rely exclusively on a perfect gossiping mechanism, which is needed for forward security for Balloon alone, as discussed in Section 5.5.

6.2.2 Forward-Secure State Generation

For the user to find his or her entries, and maintain unlinkability of entries, identifiers for entries need to be unlinkable and deterministically generated for each user in a forward secure manner. To provide deletion-detection forward integrity for individual users as well, there is also the need to authenticate all entries entirely up to a certain point in time in a forward secure manner.

The controller will keep *state* for each user, consisting of a key k and a value v that is continuously evolved by overwriting the past key and value as new entries e_i^j

for user j are generated. The initial key, k_0, is agreed upon at user registration. The key is used to generate the user-specific entry identifiers. It is constantly evolved, with each user-specific entry e_i^j, using a simple forward-secure sequential key generator (forward-secure SKG) in the form of an evolving hash-chain [19, 76, 104]:

$$k_i = \begin{cases} \texttt{Hash}(k_{i-1}), & \text{if } i > 0 \\ k_0, & \text{if } i = 0 \end{cases} \tag{6.1}$$

For deletion-detection forward integrity, we use a forward-secure sequential aggregate (FssAgg) authenticator by Ma and Tsudik [75] in the form of combining an evolving hash-chain and a MAC. The forward-secure SKG, k_i, is used in the FssAgg as follows to compute the value v_i:

$$v_i = \begin{cases} \texttt{Hash}\big(v_{i-1} || \texttt{MAC}_{k_{i-1}}(e_{i-1}^j)\big), & \text{if } i > 0 \\ v_0, & \text{if } i = 0 \end{cases} \tag{6.2}$$

We set a random v_0 of the same length as k_0 (the length of the output of the hash function) as the initial authenticator primarily to prevent a length distinguisher for one of our protocols, as described in Section 6.3.3.

6.2.3 Public-Key Encryption Scheme

Messages stored for users are encrypted under their respective public keys. The encryption scheme must provide indistinguishably under adaptive chosen ciphertext attack (IND-CCA2) [17], and key privacy under adaptive chosen ciphertext attack (IK-CCA) [16]. IK-CCA is needed for ciphertexts contained within entries, to avoid that the user an entry is generated for can be deduced from the ciphertext. IND-CCA2 is needed since our publicly verifiable proofs of message reveals the decryption of ciphertexts. This can be seen as a decryption oracle.

A publicly verifiable proof of message is in essence a proof that a given ciphertext corresponds with a given plaintext (message). These proofs can be generated by either the controller or the user, as described in Section 6.3.4. The public-key encryption scheme should enable the proofs to not require the user or the controller to reveal any long term private or secret keys, and it should provide forward security at the controller side. Only at the time of encryption (creating an entry) should the controller be able to store additional information that allows it to recover the plaintext at a later point in time. If the controller opts not to, then it should be unable to recover

any other plaintext for which it did not store this additional information which was generated during the encryption.

6.2.4 Cryptographic Primitives

The cryptographic primitives needed for a Balloon are a hash function and a signature scheme. For the forward-secure state generation, we also need a MAC scheme. Finally, we need a public-key encryption scheme.

To ease implementation, Insynd is designed around the use of NaCl [25]. The NaCl library provides all of the core operations needed to build higher-level cryptographic tools and provides state-of-the-art-security (also taking into account side-channels by having no data dependent branches, array indices or dynamic memory allocation) and high speed implementations. We selected the following primitives:

SHA-512 A collision and pre-image resistant hash function, denoted `Hash`.

Ed25519 An existentially unforgeable under chosen-message attack signature algorithm, denoted `Sign` [24].

Poly1305 A one-time existentially unforgeable message authentication code algorithm, denoted `MAC` [21].

crypto_box A public-key authenticated encryption scheme using nonces composed of Curve25519 [22], XSalsa20 [23], and Poly1305 [21]. Designed to provide privacy and third-party unforgeability in the public-key authenticated encryption setting [7].

We now go into the details of crypto_box and how we intend to use it, using the high-level functions provided by the library. Note that the encryption makes use of elliptic curve Diffie-Hellman (ECDH) and a nonce to derive a symmetric key for the subsequent symmetric authenticated encryption of the message. Three functions make up `crypto_box`:

- $\texttt{pk} \leftarrow \texttt{crypto\_box\_keypair}(\texttt{sk})$: Generates a public key `pk` and private key `sk` such that $\texttt{crypto\_scalarmult\_base}(\texttt{pk}, \texttt{sk})$, which multiplies the scalar `sk` with the Curve25519 basepoint resulting in `pk`.
- $c \leftarrow \texttt{crypto\_box}(m, n, \texttt{pk}, \texttt{sk})$: Authenticated encryption of a message m into ciphertext c using a unique nonce n, the recipient's public key `pk` and the sender's private key `sk`.

- $m \leftarrow \texttt{crypto\_box\_open}(c, n, \texttt{pk}, \texttt{sk})$: Authenticated decryption of the ciphertext c using nonce n, the sender's public key $\texttt{pk}$ and the recipient's private key $\texttt{sk}$ to learn message m. Note that $m \stackrel{?}{=} \perp$ if the decryption fails.

We will not use long-term keys for the controller (the sender key-pair in NaCl) due to our publicly verifiable proofs of message construction and the controller being forward secure. For each message that the controller wants to encrypt for a given user (the recipient key-pair in NaCl), it will generate a new ephemeral key-pair $\texttt{pk}' \leftarrow \texttt{crypto\_box\_keypair}(\texttt{sk}')$. By revealing $\texttt{sk}'$, anyone can compute $\texttt{pk}'$ using `crypto_scalarmult_base`. Note that this later plays a crucial role for publicly verifiable proofs of message. For the user to be able to perform proofs of message, it needs to know the ephemeral $\texttt{sk}'$[3]. Therefore we will simply append $\texttt{sk}'$ to each message we encrypt. This construction provides forward security at the controller side, since the controller needs to store $\texttt{sk}'$ to be able to recover the plaintext afterwards. Note that `crypto_box` requires the nonce n to be unique for a given pair of sender and receiver key-pairs, which means that we could always use an empty nonce since we create an ephemeral key-pair for each encryption. However, we use the nonce to associate the ciphertext to a given entry and to provide freshness in a protocol (both described later in Section 6.3). If not specified, the nonce is empty. Apart from the ciphertext c, the ephemeral public key $\texttt{pk}'$ also needs to be available to decrypt c. We only store a nonce as part of proofs of message. We define the following algorithms for encryption and decryption using `crypto_box`:

- $\{c, \texttt{pk}'\} \leftarrow \texttt{Enc}^{n}_{\texttt{pk}}(m)$: Encrypts a message m using an ephemeral key-pair $\texttt{pk}' \leftarrow \texttt{crypto\_box\_keypair}(\texttt{sk}')$, the public key $\texttt{pk}$, and the nonce n where the ciphertext $c = \texttt{crypto\_box}(m||\texttt{sk}', n, \texttt{pk}, \texttt{sk}')$. Returns $\{c, \texttt{pk}'\}$.

- $\{m, \texttt{sk}'\} \leftarrow \texttt{Dec}^{n}_{\texttt{sk}}(c, \texttt{pk}')$: Decrypts ciphertext c using the private key $\texttt{sk}$, public key $\texttt{pk}'$, and nonce n where $p \leftarrow \texttt{crypto\_box\_open}(c, n, \texttt{pk}', \texttt{sk})$. Note that $p \stackrel{?}{=} \perp$ if the decryption fails, otherwise $p \stackrel{?}{=} m||\texttt{sk}'$. Returns p.

- $m \leftarrow \texttt{Dec}^{n}_{\texttt{sk}',\texttt{pk}}(c, \texttt{pk}')$: Decrypts ciphertext c using the private key $\texttt{sk}'$, public key $\texttt{pk}$, and the nonce n where $p \leftarrow \texttt{crypto\_box\_open}(c, n, \texttt{pk}, \texttt{sk}')$. Note that $p \stackrel{?}{=} \perp$ if the decryption fails, otherwise $p \stackrel{?}{=} m||\texttt{sk}^*$. If $\texttt{sk}' \stackrel{?}{=} \texttt{sk}^*$ and $\texttt{pk}' \stackrel{?}{=} \texttt{pk}^*$, where $\texttt{crypto\_scalarmult\_base}(\texttt{pk}^*, \texttt{sk}')$, returns m, otherwise $\perp$.

[3] By default, the user only knows its own private key. While this is enough for decryption given the ephemeral public key, it does not reveal the ephemeral private key.

The first two algorithms enable public-key encryption and decryption with *symmetric* authenticated encryption. Since we generate a random key-pair for encryption the scheme does not retain any meaningful *public-key* authenticated encryption on its own. The third algorithm is used for publicly verifiable proofs of message, described further later in Section 6.3.4. Our construction is basically identical when it comes to encryption and decryption to the DHETM (Diffie-Hellman based Encrypt-then-MAC) construction by An [7]. The differences are that we use Curve25519, instead of a random DH group, and the symmetric authenticated key cipher based on `crypto_secretbox` as part of NaCl (based on XSalsa20 and Poly1305). Note that DHETM is provably IND-CCA2 secure.

6.3 The Insynd Scheme

Figure 6.1 shows the five protocols that make up Insynd between a controller $\mathtt{C}$, a server $\mathtt{S}$, and a user $\mathtt{U}$. The protocols are `setup` (pink box), `register` (blue box), `insert` (yellow box), `getEntry` (red box), and `getState` (green box). The following subsections describe each protocol in detail and present relevant algorithms.

6.3.1 Setup and Registration

The controller and server each have signature key-pairs, $\{\mathtt{C}_{\mathtt{sk}}, \mathtt{C}_{\mathtt{vk}}\}$ and $\{\mathtt{S}_{\mathtt{sk}}, \mathtt{S}_{\mathtt{vk}}\}$, respectively. For the controller, the key-pair is setup using `B.genkey` for Balloon. We assume that $\mathtt{C}_{\mathtt{vk}}$ and $\mathtt{S}_{\mathtt{vk}}$ are publicly attributable to the respective entities, e.g., by the use of some trustworthy public-key infrastructure. Note that, since we assume that the controller may issue new signing keys as a consequence of becoming compromised, $\{\mathtt{C}_{\mathtt{sk}}, \mathtt{C}_{\mathtt{vk}}\}$ may change over time. We consider the exact mechanism for revocation and re-issuance to be out of scope. We require that the function Ω that orders entries in Balloon, as part of $\mathtt{C}_{\mathtt{vk}}$, is constant.

Controller-Server Setup

The purpose of the `setup` protocol (pink box in Figure 6.1) is for the controller and the server to create a new Balloon, stored at the server, with two associated uniform resource identifiers (URIs). The protocol is started by the controller who sends $\mathtt{C}_{\mathtt{URI}}$ to the server. $\mathtt{C}_{\mathtt{URI}}$ specifies the URI of the state associated with the to-be created Balloon at the controller. This parameter plays a crucial role, since it will later be used by the user to query his or her state.

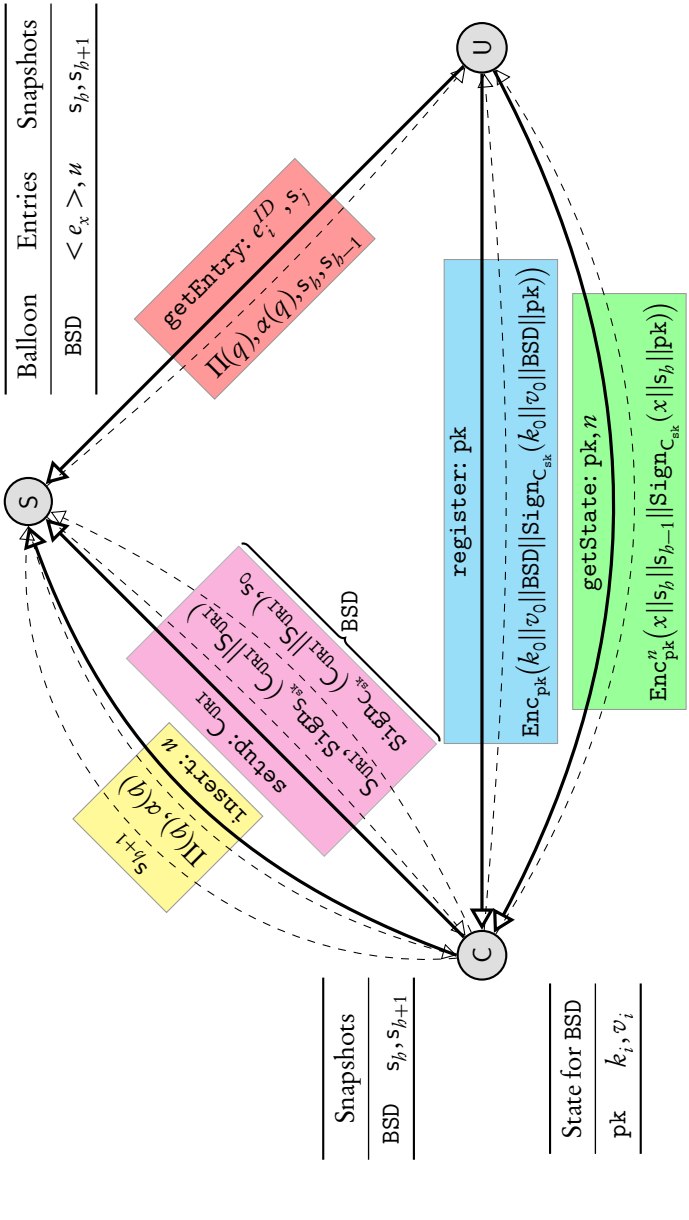


Figure 6.1 The Insynd scheme, consisting of five protocols (coloured boxes), between a controller C, a server S, and a user U. A solid line indicates the start of protocol and a dashed line a response. Section 6.3 describes each protocol in detail.

After receiving the URI that initiates the `setup` protocol, the server verifies that $\mathtt{C}_{\mathtt{URI}}$ identifies a resource under the control of the controller. If so, the server signs the URI together with the URI to the new Balloon at the server, $\mathtt{S}_{\mathtt{URI}}$. The server replies with $\mathtt{S}_{\mathtt{URI}}$ and $\mathtt{Sign}_{\mathtt{S}_{\mathtt{sk}}}(\mathtt{C}_{\mathtt{URI}}||\mathtt{S}_{\mathtt{URI}})$. The signature commits the server to the specified Balloon.

Upon receiving the reply, the controller verifies the signature and $\mathtt{S}_{\mathtt{URI}}$ from the server, and also signs two URIs. Next, the controller creates an empty Balloon $\{\mathtt{auth}(D_0), \mathtt{s}_0\} \leftarrow \mathtt{B.setup}(D_0, \mathtt{C}_{\mathtt{sk}}, \mathtt{C}_{\mathtt{vk}})$ with an empty data structure D_0. The final signature and the initial snapshot $\mathtt{s}_0$ are sent to the server to acknowledge that the new Balloon is now setup. We refer to the two signatures, two URIs, and the initial snapshot $\mathtt{s}_0$ generated as part of the `setup` protocol as the *Balloon setup data* (`BSD`). `BSD` commits both the controller and the server to the newly created Balloon, acts as an identifier for the run of the `setup` protocol, and is later used by the user for reconstruction. Once the server receives the final signature and initial snapshot, it constructs `BSD` on its own and now accepts both the `insert` and `getEntry` protocols for the Balloon[4].

User Registration

The purpose of the `register` protocol (blue box in Figure 6.1) is to enable the controller to send messages to the user, and in the process have the controller commit to how these messages will be delivered to the user. The signature (that commits the controller) is necessary to prevent the controller from fully refuting that there should exist any messages, just like in the secure logging area for FssAgg schemes as noted by Ma and Tsudik [75]. The user generates a new key-pair, $\mathtt{pk} \leftarrow \mathtt{crypto\_box\_keypair(sk)}$, and sends the public key to the controller to initiate the protocol.

Upon receiving the public key, the controller verifies that the key is 32 bytes long (all 32-byte strings are valid Curve25519 public keys for our use-case [22]), and generates an initial authentication key $k_0 \leftarrow \mathtt{Rand}(|\mathtt{Hash}(\cdot)|)$ and authenticator value $v_0 \leftarrow \mathtt{Rand}(|\mathtt{Hash}(\cdot)|)$. We generate a random v_0 to make a newly registered user in state indistinguishable from a user that has had one or more entries created for him or her. Without this, e.g., the reply from the `getState` protocol (see Section 6.3.3) would have distinguishable length. Next, the initial authentication key and authenticator value are associated with the public key `pk` of the user in the controller's *state table* for `BSD`. The state table contains the *current* authentication key k_i and authenticator value v_i for each user's public key registered in the Balloon for `BSD`. This is used

[4]Note that the server can create D_0 and $\mathtt{auth}(D_0)$ on its own based only on $\mathtt{s}_0$.

for our combined SKG and FssAgg, as described in Section 6.2.2. As a reply to the user, the controller returns k_0, v_0, the Balloon setup data BSD, and a signature by the controller: $\texttt{Sign}_{C_{sk}}(k_0||v_0||\texttt{BSD}||\texttt{pk})$. The signature also covers the public key of the user to bind the registration to a particular public key. Lastly, before sending the reply, the controller encrypts the reply with the provided public key. The encryption and signature plays an important in preventing a compromised passive adversary from learning $\{k_0, v_0\}$ when extending the registration to another controller, as described in Section 6.3.2.

On receiving the reply, the user decrypts the reply, verifies all four signatures (three in BSD), and stores the decrypted reply. It now has everything it needs to run the `getEntry` protocol, but first, we need to generate entries.

6.3.2 Generating Entries

Before describing the `insert` protocol, we need to define an entry. An entry e consists of an identifier and a payload. The identifier, e^{ID}, identifies the entry in a Balloon and is used by a user to retrieve his or her entries. The entry payload, e^{P}, contains the encrypted message from the controller. This means that the entry identifier and payload in Insynd correspond to the entry key and value in Balloon.

Algorithm 3 describes how an entry is generated by the controller. First, the controller derives a nonce n and an entry key k' from the user's current authentication key k (step 1). The current authentication key is stored in the controller's state table. The first hash prefixes a 1 to distinguish the generation of the nonce with the update of the authentication key in step 5. The structure of generating the nonce and entry key is used for publicly verifiable proofs of message, see Section 6.3.4. Figure 6.2 visualises the key derivation. In step 2, the entry identifier is generated by computing a MAC on the user's public key using the entry key. This links the entry to a particular user, which can be used for publicly verifiable proofs of user, see Section 6.3.4. In step 3, the message is encrypted using the user's public key and the generated nonce, linking the entry identifier and entry payload together. In step 4, the authenticator value v for the user aggregates the entire entry, using the construction from Section 6.2.2, and overwrites the current authenticator in state for the user. Finally, in step 5, the current authentication key is evolved using a hash function, overwriting the old value in the state table.

Algorithm 3 Generate an entry for a user.

Require: A message m, a user's public key $\texttt{pk}$ and the associated authentication key k and value v.
Ensure: An entry e and the user's state has been updated.

1: $k' \leftarrow \texttt{Hash}(n), n \leftarrow \texttt{Hash}(1||k)$
2: $e^{ID} \leftarrow \texttt{MAC}_{k'}(\texttt{pk})$
3: $e^{P} \leftarrow \texttt{Enc}^{n}_{\texttt{pk}}(m)$
4: $v \leftarrow \texttt{Hash}\big(v||\texttt{MAC}_{k}(e)\big)$
5: $k \leftarrow \texttt{Hash}(k)$
6: **return** e

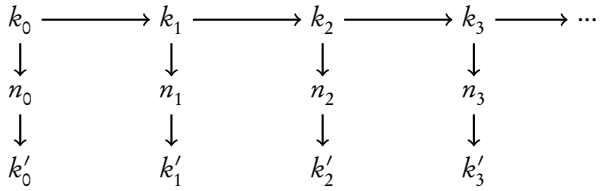


Figure 6.2 How we derive the nonce n and entry key k' from the authentication key k for entry generation.

Insert

The purpose of the `insert` protocol (yellow box in Figure 6.1) is to insert a set of entries u into a Balloon kept by the server. The server generates one or more entries using Algorithm 3 and sends u to the server at $\texttt{S}_{\texttt{URI}}$ to initiate the protocol. The server uses $\{\Pi(q), \alpha(q)\} \leftarrow \texttt{B.query}(q, D_b, \texttt{auth}(D_b), \texttt{C}_{\texttt{vk}})$ (**Prune**), described in Section 6.2.1, to generate a proof $\Pi(q)$ and answer $\alpha(q)$ for $q = u$. The server replies with $\{\Pi(q), \alpha(q)\}$.

The controller in turn uses $\{\texttt{accept}, \texttt{reject}\} \leftarrow \texttt{B.verify}(q, \alpha, \Pi, \texttt{s}_b, \texttt{C}_{\texttt{vk}})$ (**Prune**) to verify the prune query, where $\texttt{s}_b$ is the latest snapshot generated by the controller. If the verification fails, the controller restarts the protocol. Next, the controller uses $\{\texttt{s}_{b+1}, \texttt{upd}\} \leftarrow \texttt{B.update*}(u, \Pi, \texttt{s}_b, \texttt{C}_{\texttt{sk}}, \texttt{C}_{\texttt{vk}})$ to create the next snapshot $\texttt{s}_{b+1}$ (also stored in `upd`). The controller stores the snapshot in its *snapshot table* for `BSD`, and sends `upd` (the snapshot) to the server.

The server verifies the snapshot and then uses $\{D_{h+1}, \texttt{auth}(D_{h+1}), \texttt{s}_{h+1}\} \leftarrow \texttt{B.refresh}(u, D_h, \texttt{auth}(D_h), \texttt{s}_h, \texttt{upd}, \texttt{C}_{\texttt{vk}})$ to update the Balloon. Finally, the server stores the snapshot $\texttt{s}_{h+1}$ and entries u in its *Balloon table* for BSD.

Snapshots and Gossiping

Inspired by CONIKS [78], we modify the snapshot construction from the specification of Balloon. CONIKS works in a closely related setting to ours and links snapshots together into a snapshot chain, as part of their work on specifying their snapshot gossiping mechanism for an authenticated data structure similar to Balloon. We define a snapshot as follows:

$$\texttt{s}_h \leftarrow \{i, c_i, r, t, \texttt{Sign}_{\texttt{sk}}(i||c_i||r||s_{h-1}||t)\} \tag{6.3}$$

The h:th version snapshot $\texttt{s}_h$ contains the latest commitment c_i on the history tree and root r of the hash treap, both as part of Balloon ($\texttt{auth}(D_h)$), to fix the entire Balloon. Note that h is an index for the number of updates to Balloon, while i is an index for the number of entries in the Balloon. The previous snapshot, $\texttt{s}_{h-1}$, is included to form a snapshot chain. Finally, an *optional* timestamp t from a trusted time-stamping authority is included both as part of the snapshot and in the signature. The timestamp must be on $\{i||c_i||r||s_{h-1}\}$. How frequently a timestamp is included in snapshots directly influences how useful proofs of time are, as described in Section 6.3.4.

Note that timestamps do not serve any other purpose than to enable publicly verifiable proofs of time in Insynd. Timestamping of snapshots *are irrelevant* for our other properties (which we show in Section 6.4), and snapshots in general only play a role for our publicly verifiable consistency property. This means that our gossip mechanism for snapshots can be relaxed. As will become apparent, we gossip the latest snapshot as part of the `getState` and `getEntry` protocols. Since snapshots are both linked and occasionally timestamped, this greatly restricts our adversary in the forward-security model. While the `getState` protocol identifies the user, the `getEntry` protocol does not. The controller and server should make all snapshots available, e.g., on their websites.

The Last Entry

If the controller wishes to no longer be able to send messages to the user, the controller creates one last entry with the message set to a stop marker M_s and the authenticator v from state. The stop marker should be globally known, and have length $|M_s| =$

$2|\texttt{Hash}(\cdot)|$ to match the length of $\{k, v\}$ in state (see Section 6.3.3). The stop marker and authenticator are necessary for enabling users to distinguish between the correct stop of sending messages and modifications by an attacker, as noted by Ma and Tsudik [75]. After the entry with the stop marker and authenticator message has been inserted, the controller can delete the entry in the BSD state table for the user. For each registered user who has not had a last entry inserted, the controller has to keep the user's entry in the state table.

Extend

The controller has the ability to *extend* a registration made by a user. Extending a registration for a user is done in three steps:

1. Given pk, generate a *blinded* public key $\texttt{pk}'$ using the blinding value b, where b is randomly generated, as in Section 3.4.5.
2. Run the `register` protocol, as described in Section 6.3.1, using $\texttt{pk}'$ to initiate the protocol.
3. Concatenate the result from step 2, together with blinding value b from step 1 and the extend marker M_e, as a message and send it as an entry to the user.

Extending a registration, as a consequence of step 2, is a protocol between two parties. The initiating controller can either run the protocol with itself, or with another controller, where the initiating controller takes on the role of the user. Running the protocol with itself serves two purposes:

- First, this introduces new randomness for the user, recovering *future entries* from the limited compromise of a passive adversary in the past.
- Secondly, this enables the controller to register the user in a new Balloon without interacting with the user.

When the initiating controller runs the protocol with another controller, then the initiating controller is in effect registering the user for receiving messages from another controller. This enables non-interactive registration in distributed settings, once the user has registered with at least one controller.

We blind the public key of the user for three reasons: first, it hides information from an adversary that compromises multiple controllers, preventing trivial correlation of state tables to determine if the same user has been registered at both controllers

(ensuring forward unlinkability of user identifiers). Secondly, Insynd uses the public key as an identifier for the registration of a user. For each registration, we need a new identifier. Last, but not least, the blinding approach (compared to, e.g., just having the controller create a new key-pair which would still fit our adversary model) has the added benefit of if run correctly it is still only the user that at any point in time has the ability to decrypt entries intended for it.

6.3.3 Reconstruction

A user uses two protocols to reconstruct his or her messages sent by the controller: `getEntry` and `getState`. We first explain how to get entries using the `getEntry` protocol, then how to verify the authenticity of the entries with the help of the `getState` protocol, and end by discussing some options for minimising information leaks during reconstruction.

Getting Entries

To download his or her i:th entry from the server for the user, the user calculates the entry identifier e_i^{ID} using k_0 and $\mathtt{pk}$ from the registration (as described in Section 6.3.1) together with equation (1) in Section 6.2.2 to calculate k_i, as follows:

$$k' = \mathtt{Hash}\big(\mathtt{Hash}(1||k_i)\big) \tag{6.4}$$

$$e_i^{ID} = \mathtt{MAC}_{k'}(\mathtt{pk}) \tag{6.5}$$

The structure of entry identifiers is determined by Algorithm 3. The Balloon setup data, `BSD`, contains the URIs to contact the server and controller at. To get an entry, the user initiates the `getEntry` protocol (red box in Figure 6.1) by sending the identifier e_i^{ID} to the server together with an optional snapshot $\mathtt{s}_j$.

The server generates a reply using the provided identifier and snapshot as q by running $\{\Pi(q), \alpha(q)\} \leftarrow \mathtt{B.query}(q, D_h, \mathtt{auth}(D_h), \mathtt{C_{vk}})$ (**Membership**). If no snapshot is provided, the server uses the latest snapshot $\mathtt{s}_h$. We want to retain the ability to query for any snapshot $\mathtt{s}_j$ for sake of proofs of time (see Section 6.3.4). The server replies with $\{\Pi(q), \alpha(q), \mathtt{s}_h, \mathtt{s}_{h-1}\}$, where $\mathtt{s}_h$ is the latest snapshot and $\mathtt{s}_{h-1}$ the snapshot prior to $\mathtt{s}_h$ (if it exists). We include the snapshots to enable the user to verify the signature on $\mathtt{s}_h$ in case it does not know all snapshots. This also acts as a gossip mechanism, using the server to distribute snapshots.

The user verifies the reply with $\{\texttt{accept},\texttt{reject}\} \leftarrow \texttt{B.verify}(q, \alpha, \Pi, \texttt{s}_h, \texttt{C}_{\texttt{vk}})$ (**Membership**) together with the snapshot format specified in Section 6.3.2. The user continues requesting entries until the server provides a non-membership proof. Next, the user verifies the authenticity of all retrieved entries.

Verifying Authenticity

The `getState` protocol (green box in Figure 6.1) plays a central role in determining the authenticity of the entries retrieved from the server. The user initiates the protocol by sending his or her public key `pk` and a nonce $n \leftarrow \texttt{Rand}(|\texttt{Hash}(\cdot)|)$ to the controller.

Upon receiving the public key and nonce, the controller validates the public key and inspects its state table for the `BSD` in question (the controller can determine which `BSD` based upon the URI at the controller, $\texttt{C}_{\texttt{URI}}$, the request was sent to) and checks if it contains an entry for `pk`. If there is an entry, the controller sets $x \leftarrow \{k_i, v_i\}$, where k_i is the current authentication key and v_i the current authenticator in the state table for `pk`. If there is not, then the controller sets $x \leftarrow M_s$. Note that M_s has the same length as $\{k_i, v_i\}$. The controller sends as its reply $\texttt{Enc}^n_{\texttt{pk}}\big(x||\texttt{s}_h||\texttt{s}_{h-1}||\texttt{Sign}_{\texttt{C}_{\texttt{sk}}}(x||\texttt{s}_h||\texttt{pk})\big)$. The reply is encrypted using the provided public key and nonce since anyone can initiate the protocol. The nonce ensures *freshness* of the reply (since it is provided by the user), which matters for security, later discussed in Section 6.4. The encryption prevents any third-party with a user's public key to determine if new entries are generated for the user based on observing the reply, since `Enc` is randomised. The reply contains x (previously set based upon the state table), the latest snapshot $\texttt{s}_h$, the next-latest snapshot $\texttt{s}_{h-1}$, and a signature. The signature covers x, the latest snapshot, and the public key. This forces the controller to commit to a particular state x for a particular public key `pk` at a particular state of the Balloon fixed by $\texttt{s}_h$. The two snapshots are included in the reply both to enable the user to verify the signature from the controller and to act as a form of gossiping mechanism. Note that this gossiping mechanism is weaker than the gossiping taking place as part of the `getEntry` protocol, since the controller presumably knows which user is performing `getState`.

The user decrypts the reply, verifies the signature and latest snapshot. If any of these operations fail, the verification of authenticity fails. With the list of entries downloaded as described in Section 6.3.3, the user can now use Algorithm 4 to decrypt all entries and check the consistency between entries and the reply from `getState`. Steps 1–8 decrypt all entries using the nonce, entry key, and authentication key generation determined by Algorithm 3. If a stop marker (M_s) is found (steps 6–7), the authenticator v' in the last entry should match the calculated authenticator v and

`getState` should have returned M_s. If no stop marker is found in an entry, then the reply from `getState` should match the calculated state in the form of $\{k, v\}$ (step 9). For the verification algorithm to correctly check consistency between entries and `getState`, the non-membership proof that caused the user to stop downloading entries must be for the same snapshot as the reply from `getState`. If the snapshots do not match, then entries may have been inserted after in subsequent snapshots causing the verification to incorrectly fail.

Algorithm 4 Verify authenticity of entries for one user.

Require: $\{\mathtt{pk}, \mathtt{sk}, k_0, v_0\}$, the reply x from `getState`, an ordered list l of entries.
Ensure: `true` if all entries are authentic and the state x is consistent with the entries in l, otherwise `false`.

1: $k' \leftarrow \mathtt{Hash}(n), n \leftarrow \mathtt{Hash}(1||k), k \leftarrow k_0, v \leftarrow v_0$ ▷ k' is the entry key, n the entry nonce, k and v the calculated state
2: **for all** $e \in l$ **do** ▷ in the order entries were inserted
3: $p \leftarrow \mathtt{Dec}^n_{\mathtt{sk}}(e^P)$
4: **if** $p \stackrel{?}{=} \perp$ **then**
5: **return** `false` ▷ failed to decrypt entry
6: **if** p contains $\{M_s, v'\}$ **then** ▷ check for M_s, v' unknown
7: **return** $x \stackrel{?}{=} M_s \wedge v \stackrel{?}{=} v'$ ▷ no state and matching authenticator
8: $k' \leftarrow \mathtt{Hash}(n), n \leftarrow \mathtt{Hash}(1||k), k \leftarrow \mathtt{Hash}(k), v \leftarrow \mathtt{Hash}\big(v||\mathtt{MAC}_k(e)\big)$ ▷ calculated from right to left
9: **return** $x \stackrel{?}{=} \{k, v\}$ ▷ state should match calculated state

For each extension marker, M_e, found in the message of a decrypted entry, the user simply reruns the reconstruction steps outlined in this section after generating the blinded private key.

Privacy-Preserving Download

By downloading his or her entries, the user inadvertently leaks information that could impact the user's privacy. For instance, the naive approach of downloading entries as fast as possible risks linking the entries together, and their relative order within each run of the `insert` protocol, due to time, despite our (assumption of) an anonymous channel of communication. To minimise information leakage, we could:

- Have users wait between requests. The waiting period should be randomly sampled from an exponential distribution, which is the maximum entropy prob-

ability distribution having mean $1/N$ with rate parameter N [87], assuming that some users may never access their entries ($[0, \infty]$).

- Randomise the order in which entries are requested. The `getState` protocol returns k_i, which enables the calculation of how many entries have been generated so far using k_o.
- Introduce request and response padding to minimise the traffic profile, e.g., to 16 KiB as is done in Pond.
- Use private information retrieval (PIR) [36, 43]. This is relatively costly but may be preferable over inducing significant waiting time between requests.
- Since servers are not assumed trusted, their contents could safely be mirrored, enabling users to spread their requests (presumably some mirrors do not collude).

For now, we opt for the first two options of adding delays to requests and randomising the order. PIR-enabled servers would provide added value to users. We note that the mirroring approach goes well in hand with verifying the consistency of snapshots, since mirrors could *monitor* snapshot generation as well as serve entries to users. We note thoroughly evaluating the trade-offs for the different options for privacy-preserving downloading as interesting future work.

6.3.4 Publicly Verifiable Proofs

Insynd allows for four types of publicly verifiable proofs: controller, time of existence, user, and message. These proofs can be combined to, at most, prove that the controller had sent a message to a user at a particular point in time. While the controller and time proofs can be generated by anyone, the proofs of user and message can only be generated by the user (always) and the controller (if it has stored additional information at the time of generating the entry).

Controller

To prove who the *controller* of a particular entry is, i.e., that a controller created an entry, we rely on Balloon. The proof is the output from `B.query` (**Membership**) for the entry. Verifying the proof uses `B.verify` (**Membership**).

Time

To prove *when* an entry *existed*. The granularity of this proof depends on the frequency of timestamped snapshots. The proof is the output from `B.query` (**Membership**) for the entry from a timestamped snapshot $\mathtt{s}_j$ that shows that the entry was part of the data structure fixed by $\mathtt{s}_j$. Verifying the proof involves using `B.verify` (**Membership**) and whatever mechanism is involved in verifying the timestamp from the time-stamping authority. The proof shows that the entry existed at the time of the timestamp.

User

To prove who the recipient *user* of a particular entry is, the proof is:

1. the output from `B.query` (**Membership**) for the entry, and
2. the entry key k' and public key `pk` used to generate the entry identifier e^{ID}.

Verifying the proof involves using `B.verify` (**Membership**), calculating $\tilde{e}^{ID} = \mathtt{MAC}_{k'}(\mathtt{pk})$ and comparing it to the entry identifier e^{ID}. The user can always generate this proof, while the controller needs to store the entry key k' and public key `pk` at the time of entry generation. If the controller stores this material, then the entry is linkable to the user's public key. If linking an entry to a user's public key is not adequately attributing an entry to a user (e.g., due to the user normally being identified by an account name), then the `register` protocol should also include an extra signature linking the public key to additional information, such as an account name.

Message

The publicly verifiable proof of message includes a publicly verifiable proof of user, which establishes that the ciphertext as part of an entry was generated for a specific public key (user). The proof is:

1. the output from `B.query` (**Membership**) for the entry,
2. the nonce n that is needed for decryption and used to derive the entry key k' (see Figure 6.2),
3. the public key `pk` used to generate e^{ID}, and
4. the ephemeral secret key $\mathtt{sk}'$ that is needed for decryption.

Verifying the proof involves first verifying the proofs of controller and user. Next, the verifier can use $\texttt{Dec}^{n}_{\texttt{sk'},\texttt{pk}}(c, \texttt{pk'})$ from Section 6.2.4 to learn the message m. The user can always generate this proof, while the controller needs to keep the nonce n, public key pk, and the ephemeral private key sk' at entry generation. Note that the controller is allowed to save key material to produce proofs of message of entries, never for the encrypted replies to the `getState` or `register` protocols.

6.4 Evaluation

We evaluate all properties in the forward security model. For recovery from a time-limited passive adversary, as described in Section 6.1, we rely on our extension messages from Section 6.3.2. Once the controller has recovered from the time-limited compromise by the passive adversary, the next extension message introduces new randomness by rerunning the `register` protocol for a user. We assume that the controller, as part of recovering from a time-limited compromise, revokes its old signature key and generates a new one that is associated to the controller. If the controller uses a Hardware Security Module (HSM) to generate its signatures, this assumption is no longer necessary.

Where possible, we make use of the model (and notation) from Chapter 3, with some modifications. The `CorruptController` oracle corresponds to the `CorruptServer` oracle. We need an oracle for the controller, and not the server, since it is the controller that holds state in Insynd. Note that, without loss of generality, we assume a single controller C and server S. For properties with a "forward" prefix, the adversary cannot make any further queries to its oracles after it makes a call to `CorruptController`. For sake of clarity, we give full definitions. The updated model for Insynd is as follows for an adversary $\mathcal{A}$ that can adaptively control the scheme through a set of oracles:

- $\texttt{pk} \leftarrow \texttt{CreateUser}()$: Uses $\texttt{pk} \leftarrow \texttt{crypto\_box\_keypair}(\texttt{sk})$ to generate a new public key pk to identify a user, registers the user pk at the controller C, and finally returns pk.
- $e \leftarrow \texttt{CreateEntry}(\texttt{pk}, m)$: Creates an entry e with message m at the controller C using Algorithm 3 for the user pk registered at C. Returns e.
- $\{\texttt{State}, \texttt{\#entries}\} \leftarrow \texttt{CorruptController}()$: Corrupt the controller C, which returns the entire state of the controller and the number of created entries before calling this oracle.

6.4.1 Secrecy

We define a modified `CreateEntry` oracle for the challenge:

- $e \leftarrow$ `CreateEntry*`$(\texttt{pk}, m_0, m_1)_b$: Creates an entry e with message m_b at the controller `C` using Algorithm 3 for the user `pk` registered at `C`. Returns e. Note that this oracle can only be called once with distinct messages m_0 and m_1 of the same length.

To model secrecy in Insynd we also need to provide the adversary with a decryption oracle. This is to model the side information the adversary can get hold off through publicly verifiable proofs of message (or possibly the stored data at the controller to be able to make these proofs). Note that the proofs of message are only allowed for entries, not replies for the `getState` or `register` protocols. The adversary can make queries to the decryption oracle $m \leftarrow$ `DecryptEntry`(e) with the restriction that it can only decrypt entries which were generated by the normal `CreateEntry` oracle. The experiment is:
$\mathbf{Exp}^{SE}_{\mathscr{A}}(k)$:

1. $b \in_R \{0,1\}$
2. $g \leftarrow \mathscr{A}^{\texttt{DecryptEntry},\texttt{CreateUser},\texttt{CreateEntry},\texttt{CreateEntry*},\texttt{CorruptController}}()$
3. Return $g \stackrel{?}{=} b$.

The advantage of the adversary is defined as

$$\mathbf{Adv}^{SE}_{\mathscr{A}}(k) = \frac{1}{2} \cdot \left| Pr\left[\mathbf{Exp}^{SE}_{\mathscr{A}}(k) = 1 | b = 0\right] + Pr\left[\mathbf{Exp}^{SE}_{\mathscr{A}}(k) = 1 | b = 1\right] - 1 \right|.$$

Based on our experiment we can now define secrecy:

Definition 10. *A scheme provides computational secrecy of messages contained within entries, if and only if for all polynomial time adversaries $\mathscr{A}$, it holds that $\mathbf{Adv}^{SE}_{\mathscr{A}}(k) \leq \epsilon(k)$.*

With secrecy defined, we can specify our secrecy theorem:

Theorem 9. *For an IND-CCA2 secure public-key encryption scheme, Insynd provides computational secrecy of the messages contained in entries, according to Definition 10.*

Proof sketch. For each encryption with `crypto_box`, we generate an ephemeral key-pair, as specified in Section 6.2.4. The ephemeral private key is what is disclosed as part of proofs of message together with a nonce. This means that the information provided in proofs of message reveals no additional information to the adversary beyond the already known public key of the user and, notably, the actual message (plaintext) in a ciphertext and the nonce. Conservatively, this means that proofs of message can be seen as a decryption oracle where the adversary provides the ciphertext, and as part of constructing the ciphertext picks a nonce. Since `crypto_box` only requires a unique nonce for a given pair of sender and receiver key-pairs, and we generate an ephemeral key-pair for each encryption, the advantage of the adversary is identical as in the IND-CCA2 game for `crypto_box` with ephemeral sender key-pairs.

□

6.4.2 Deletion-Detection Forward Integrity

We define an algorithm $\texttt{valid}(e,i)$ that returns whether or not the full log trail for every user verifies when e_i (the entry e created at the i-th call of `CreateEntry`) is replaced by e. Since we do not assume the server to be trusted, we may view it as if the adversary is always in control of the server. We therefore take the conservative approach of specifying a new oracle:

- $c \leftarrow \texttt{GetState}(\texttt{pk}, n)$: Returns the ciphertext c that is generated as the reply to the `getState` protocol for the user `pk` and nonce n.

We define the experiment for forward integrity (FI) as:
$\mathbf{Exp}^{FI}_{\mathscr{A}}(k)$:

1. $l \leftarrow \mathscr{A}^{\texttt{CreateUser},\texttt{CreateEntry},\texttt{CorruptController},\texttt{GetState}}()$
2. Return $e \neq e_i \wedge \texttt{valid}(e,i) \wedge i \leq \texttt{\#entries}$.

The advantage of the adversary is defined as

$$\mathbf{Adv}^{FI}_{\mathscr{A}}(k) = Pr\left[\mathbf{Exp}^{FI}_{\mathscr{A}}(k) = 1\right].$$

Based on our experiment we can now define forward integrity and deletion-detection forward integrity:

Definition 11. *A scheme provides computational forward integrity, if and only if for all polynomial time adversaries $\mathscr{A}$, it holds that $\mathbf{Adv}^{FI}_{\mathscr{A}}(k) \leq \epsilon(k)$.*

Definition 12. *A scheme provides computational deletion-detection forward integrity, if and only if it is FI secure and the verifier can determine, given the output of the scheme and the time of compromise, whether any prior entries have been deleted.*

With deletion-detection forward integrity defined, we can specify the deletion-detection forward integrity theorem for Insynd:

Theorem 10. *Given an unforgeable signature algorithm, an unforgeable one-time MAC, and a IND-CCA2 secure public-key encryption algorithm, Insynd provides computational deletion-detection forward integrity in the random oracle model, according to Definition 12.*

Proof sketch. Insynd provides deletion-detection forward integrity for user's entries primarily thanks to the use of the FssAgg authenticator by Ma and Tsudik [75] which is provably secure in the random oracle model. The verification is "all or nothing", i.e., we can detect if all entries are authentic or not, not which entries have been tampered with. We look at three distinct phases of a user in Insynd: before registration, after registration, and after the last entry.

Before registration, the controller has not generated any entries for the user that can be modified or deleted. The `register` protocol establishes the initial key and value for the forward-secure SKG and FssAgg authenticator, as described in Sections 6.2.2 and 6.3.1. Furthermore, the controller signs the initial key and value together with the `BSD` and public key of user. This forces the controller to commit to the existence of either *state* (in the form of the initial key and value) or at least one entry for the user (the last entry with the stop marker and authenticator) assuming an unforgeable signature algorithm. These steps correspond to the log file initialization in the Ma and Tsudik construct.

After registration, but before the last entry, the controller must keep state for the user. The state is in the form of the current key k and authenticator v. The user gets the current state using the `getState` protocol for his or her public key and a fresh nonce. The reply from the controller is encrypted under the user's provided public key and the nonce provided by the user. This ensures the *freshness* of the reply, preventing the adversary from caching replies from the `getState` protocol made prior to compromise of the controller (using the `GetState` oracle). In Algorithm 3, steps 4–5 authenticate the entire entry into v and overwrite the key k. This construction is identical to the Ma and Tsudik privately verifiable FssAgg construction that is provable forward-secure given an unforgeable MAC in the random oracle model.

Note that each entry uses a unique key for the MAC, where the key is derived from a hash function for which the adversary does not know the input.

After the last entry, the controller has deleted state. The last entry, as described in Section 6.3.2, contains a stop marker M_s and the authenticator v before the last entry was generated. The verification algorithm, Algorithm 4, verifies the authenticator in the case of state (step 9) and no state (step 7). Compared to the privately verifiable FssAgg construction, we store the authenticator in an entry instead of state. The important thing is that we still verify the authenticator, and fail if we are unable to (steps 4 and 7).

Finally, we note that the use of an IND-CCA2 secure encryption scheme prevents an adversary from learning any authenticator keys and values from the `GetState` oracle.

□

6.4.3 Forward Unlinkability of Entries

It should be hard for an adversary to determine whether or not a relationship exists between two entries in one round of the `insert` protocol, even when given at the end of the round the entire state of the controller: $\{\mathtt{pk}, k, v\}$ of all users. We also need to take into account that for certain entries, publicly verifiable proofs of user or message (or data stored at the controller to be able to make these proofs) are available to the adversary. Obviously, the adversary knows the user of these messages, and does not break forward unlinkability of entries, if it can only establish a link between two entries in one round of the insert protocol for which it knows the user.

Even though we do not define forward unlinkability of entries in terms of unlinkability between entries and users, it can be modelled as in Chapter 3 for forward unlinkability of entries, where the adversary can only create users before each round of the `insert` protocol and is limited to finding a relationship between two entries within one round. In the definition from Section 3.3, the adversary has to output a guess for the bit b for an entry created for one out of two users (which user is specified by the bit). Note, however, that this definition also covers that it is hard for the adversary to link a user to one particular entry. For the following sequence of oracles calls:

1. $vuser \leftarrow \mathtt{Drawuser}(\mathrm{A},\mathrm{B})$

2. $e_1 \leftarrow \mathtt{CreateEntry}_b(vuser, m)$

3. `Free`($vuser$)
4. $vuser \leftarrow$ `Drawuser`(A,C)
5. $e_2 \leftarrow$ `CreateEntry`$_b$($vuser, m$)
6. `Free`($vuser$)

Finding a relationship between e_1 and e_2 implies that the bit b is `0`. The entries for which publicly verifiable proofs of user or message are available (or for which the controller stored data to be able to make these proofs) can be modelled by creating these entries by invoking the `CreateEntry` oracle. This means we can use the approach from Section 3.3.4 also for forward unlinkability of entries. We reuse the `DrawUser` and `Free` oracles. In addition we define a new `CreateEntry'` oracle:

- $e \leftarrow$ `CreateEntry'`($vuser, m$)$_b$: Creates an entry e with message m at the controller `C` using Algorithm 3 for a user registered at `C`. Which user depends on the value b and the entry $vuser$ in the table $\mathscr{D}$. Returns e.

We define the experiment for forward unlinkability of entries as:
$\mathbf{Exp}^{FU}_{\mathscr{A}}(k)$:

1. $b \in_R \{0,1\}$
2. $g \leftarrow \mathscr{A}^{\texttt{CreateUser,DrawUser,Free,CreateEntry,CreateEntry',GetState,CorruptController}}()$
3. Return $g \stackrel{?}{=} b$.

The advantage of the adversary is defined as

$$\mathbf{Adv}^{FU}_{\mathscr{A}}(k) = \frac{1}{2} \cdot \left| Pr\left[\mathbf{Exp}^{FU}_{\mathscr{A}}(k) = 1 | b = 0\right] + Pr\left[\mathbf{Exp}^{FU}_{\mathscr{A}}(k) = 1 | b = 1\right] - 1 \right|.$$

With the experiment in place, we can define forward unlinkability of entries and the associated theorem for Insynd:

Definition 13. *A scheme provides computational forward unlinkability of entries, if and only if for all polynomial time adversaries $\mathscr{A}$, it holds that $\mathbf{Adv}^{FU}_{\mathscr{A}}(k) \leq \epsilon(k)$.*

Theorem 11. *For an IND-CCA2 secure public-key encryption algorithm, Insynd provides computational forward unlinkability of entries within one round of the* `insert` *protocol in the random oracle model according to Definition 13.*

Proof sketch. The adversary has access to the following information for entries for which no publicly verifiable proofs of user or message (or the data stored at the controller to be able to create these) are available: $e^{ID} = \texttt{MAC}_{k'}(\texttt{pk})$ and $e^{P} = \texttt{Enc}^{n}_{\texttt{pk}}(m)$ for which $k' = \texttt{Hash}(n)$ and $n = \texttt{Hash}(1||k)$ where k is the current key for the user at the time of generating the entry.

In the random oracle model, the output of a hash function is replaced by a random value. Note that the output of the random oracle remains the same for the same input. By assuming the random oracle model, the key to the one-time unforgeable MAC function and the nonce as input of the encryption are truly random[5]. Hence the adversary that does not know the inputs of these hashes, n and k respectively, has no advantage towards winning the indistinguishability game.

Now we need to show that the adversary will not learn these values n and k, even when given the controller's entire state $\{\texttt{pk}, k, v\}$ for all active users and values k' and n for entries where a publicly verifiable proof of user or message is available (or can be constructed from data the controller stored additionally at the time of generating these entries). The state variable k is generated using a forward-secure sequential key generator in the form of an evolving hash chain. There is no direct link between the values n, respectively k', of multiple entries for the same user. Instead, for each entry these values are derived from the user's current authentication key k at that time, using a random oracle. The adversary can thus not learn the value of the past authentication keys from the values n and k'. Lastly, from state, the adversary learns pk for all users. We note that the encryption for entries provides key privacy as assumed in Section 6.2.4, because the outputted ciphertext is encrypted using a symmetric key that is derived from both the Diffie-Hellman value and the nonce. Even when assuming that the adversary can compute the Diffie-Hellman value, it has no information on the nonce and hence the encryption provides key privacy, i.e., one cannot tell that the ciphertext was generated for a given public key.

We need to show that the adversary will not be able to link entries together from the state variable v it keeps for every user. If $v \stackrel{?}{=} v_0$, then v is random. Otherwise, $v_i = \texttt{Hash}\big(v_{i-1}||\texttt{MAC}_{k_{i-1}}(e_{i-1})\big)$. The MAC is keyed with the previous authentication key k_{i-1}, which is either the output of a random oracle (if $i > 1$) or random (k_0). This means the adversary does not know the output of $\texttt{MAC}_{k_{i-1}}(e^{j}_{i-1})$ that is part of the input for the random oracle to generate v.

[5]If the adversary has no knowledge of n, it cannot tell whether or not a given ciphertext was encrypted for a given user (with known public key), even when given the ephemeral secret key sk′ used to seal crypto_box. This implies that the adversary will also not gain any advantage from learning the corresponding ephemeral public key pk′, as part of the ciphertext.

Finally, we note that the use of an IND-CCA2 secure encryption scheme prevents an adversary from learning any authenticator keys and values from the `GetState` oracle.

□

6.4.4 Publicly Verifiable Consistency

This follows directly from the properties of Balloon (see Theorem 8 in Section 5.5), assuming a collision resistant hash function, an unforgeable signature algorithm, monitors, and a perfect gossiping mechanism for snapshots. For Insynd, our gossiping mechanisms are imperfect. We rely on the fact that (1) users can detect any modifications on their own entries and (2) snapshots are chained together and occasionally timestamped, to deter the controller from creating inconsistent snapshots. If stronger guarantees are needed, Insynd needs a better gossiping mechanism.

6.4.5 Publicly Verifiable Proofs

We look at each proof individually, arguing for why they cannot be trivially forged. For proofs of controller, we rely on an unforgeable signature algorithm and a collision-resistant hash function. Proofs of time rely on a proof of controller and the underlying time-stamping mechanism. Proofs of user rely on a proof of controller and an unforgeable MAC. Finally, proofs of message rely on a proof of controller, a proof of user, a pre-image resistant hash function, the structure of Curve25519, and that `crypto_box_open` is deterministic.

Controller

A proof of controller for an entry is the output from `B.query` (**Membership**) for the entry. First, the signature in the snapshot in the query cannot be forged, since the signature algorithm is assumed to be unforgeable. In Chapter 5, Theorem 7 proves the security of the membership query for Balloon assuming the collision-resistance of the underlying hash function. However, in this definition of security, the adversary only has oracle access to producing new snapshots. In the forward security model, the adversary learns the private signing key, and can therefore create arbitrary snapshots on its own. We note that the signature algorithm is still unforgeable, and therefore the existence of a signature is non-repudiable evidence of the snapshot having been made with the signing key. If the signing key has been revoked or not we consider out of scope.

Time

A proof of time for an entry consists of a proof of controller from a snapshot that contains a time-stamp from a time-stamping authority. Forging a proof of controller involves, as noted previously, forging the signature on the snapshot or breaking the collision-resistance of the underlying hash function. In addition, a proof of time depends on the time-stamping mechanism, which we consider out of scope. We note that a proof of time only proves that an entry existed at a particular point in time as indicated by the time-stamp, not that the entry was inserted or generated at that point in time.

User

A proof of user (entry recipient) consists of a proof of controller, a public key $\texttt{pk}$, and an entry key k'. Forging a proof of controller involves, as noted previously, forging the signature on the snapshot or breaking the collision-resistance of the underlying hash function. The proof of controller fixes the entry, which consists of an entry identifier e^{ID} and an entry payload. The proof is verified by computing and comparing $e^{ID} \stackrel{?}{=} \texttt{MAC}_{k'}(\texttt{pk})$. Forging the entry identifier is therefore forging the tag of the MAC since the tag is fixed by the proof of controller, which is not possible, since the MAC is unforgeable.

Message

A proof of message consists of a proof of controller, a public key $\texttt{pk}$, a nonce n, and an ephemeral secret key $\texttt{sk}'$. The proof of controller, public key, and entry key $k' \leftarrow \texttt{Hash}(n)$ (see Algorithm 3) results in a proof of user. The proof of controller fixes the entry that consists of the entry identifier and entry payload. The proof of user fixes the public key, entry key and nonce, since the prover provided a pre-image of the entry key (the nonce). The payload consists of the ciphertext c and the ephemeral public key $\texttt{pk}'$. The prover provides $\texttt{sk}'$, such that $\texttt{pk}' \stackrel{?}{=} \texttt{pk}^*$, where $\texttt{crypto\_scalarmult\_base}(\texttt{pk}^*, \texttt{sk}')$. This fixes $\texttt{sk}'$, since there is only one $\texttt{sk}'$ for each $\texttt{pk}'$ for Curve25519[6]. This fixes all the input to `crypto_box_open`: $c, n, \texttt{pk}$ and $\texttt{sk}'$, and `crypto_box_open` is deterministic.

[6]There are two points on the curve for Curve25519 such that `crypto_scalarmult_base(pk', sk')` due to Curve25519 being a Montgomery curve, but Curve25519 only operates on the x-coordinate [22].

6.4.6 Non-Interactive Registration

Our extension messages, as described in Section 6.3.2, do not require the user to interact. Blinded user identifiers are unlinkable assuming that the DDH assumption is valid for Curve25519. This is similar to the support for distributed settings and forward-unlinkability of user identifiers in Section 3.3.2.

6.5 Related Work

Ma and Tsudik also propose a publicly verifiable FssAgg scheme by using an efficient aggregate signature scheme, BLS [29, 75]. The main drawbacks are a linear number of verification keys with the number of runs of the key update, and relative expensive bilinear map operations. Similarly, Logcrypt by Holt [64] also needs a linear number of verification keys with key updates. The efficient public verifiability, of both the entire Balloon and individual entries, of Insynd comes from taking the same approach as (and building upon) the History Tree system by Crosby and Wallach [38] based on authenticated data structures. The main drawback of the work of Crosby and Wallach, and to a lesser degree of ours on Insynd, is the reliance upon a gossiping mechanism. Insynd takes the best of both worlds: the public verifiability from authenticated data structures based on Merkle trees, and the private all-or-nothing verifiability of the privately verifiable FssAgg scheme from the secure logging area. Users do not have to rely on perfect gossiping of snapshots, while the existence of private verifiability for users deters an adversary from abusing the lack of a perfect gossiping mechanism to begin with. This is similar to the approach of CONIKS [78].

PillarBox is a fast forward-secure logging system by Bowers *et al.* [31]. Beyond integrity protection, PillarBox also provides a property referred to as "stealth" that prevents a forward-secure adversary from distinguishing if any messages are inside an encapsulated buffer or not. This indistinguishability property is similar to our forward unlinkability of entries property. PillarBox has also been designed to be fast with regard to securing logged messages. The goal is to minimise the probability that an adversary that compromises a system will be able to shut down PillarBox before the entries that (presumably) were generated as a consequence of the adversary compromising the system are secured.

Pond[7] and WhisperSystem's TextSecure[8] are prime examples of related secure asynchronous messaging systems. While these systems are for two-way commu-

[7] https://pond.imperialviolet.org, accessed 2015-01-29.
[8] https://whispersystems.org, accessed 2015-01-29.

nication, there are several similarities. Pond, like Insynd, relies on an anonymity network to function. Both Pond and TextSecure introduce dedicated servers for storing encrypted messages. In Pond, clients pull their server for new messages with exponentially distributed connections. The Axolotl ratchet[9] is used by both Pond and TextSecure. Axolotl is inspired by the Off-the-Record Messaging protocol [30] and provides among other things forward secrecy and recovery from compromise of keys by a passive adversary. Our extension messages, Section 6.3.2, mimic the behaviour in Axolotl, in the sense that new ephemeral keying material is sent with messages. Note that the goal of Insynd is for messages to be non-repudiable, unlike Pond, TextSecure and OTR who specifically want *deniability*. Insynd achieves non-repudiation through the use of Balloon and how we encrypt messages, as outlined for proofs of messages in Section 6.3.4.

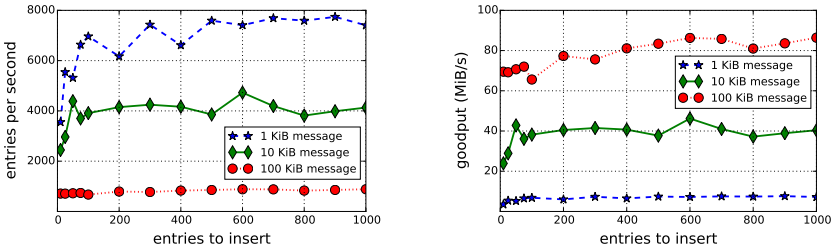
6.6 Performance

We implemented Insynd in the Go programming language. The source code and steps to reproduce our benchmark are available at http://www.cs.kau.se/pulls/insynd/. We performed our benchmark on Debian 7.8 (x64) using an Intel i5-3320M quad core 2.6GHz CPU and 7.7 GB DDR3 RAM. The performance benchmark focuses on the `insert` protocol since the other protocols are relatively infrequently used, and reconstruction time (as described in Section 6.3.3) is presumably dominated by the mechanism used to protect privacy when downloading entries. For our benchmark, we ran the controller and server on the same machine but still generated and verified proofs of correct insertion into the Balloon.

Figure 6.3 presents our benchmark, based on averages after 10 runs using Go's built-in benchmarking tool, where the x-axis specifies the number of entries to insert per run of the `insert` protocol. Figure 6.3a shows the number of entries per second for different message sizes in a Balloon of 2^{20} entries. Clearly, the smaller the message the more entries can be sent per second. We get ≈7000 entries per second with 1 KiB messages when inserting at least 100 entries per run. Using the same data as in Figure 6.3a, Figure 6.3b shows the goodput (not including entry overhead of 112 bytes per entry) for the different message sizes. At ≈800 100 KiB-messages per second (around at least 200 entries to insert), the goodput is ≈80 MiB/s. 10 KiB messages offer a trade-off between goodput and number of entries, providing 4000 entries per second with ≈40 MiB/s goodput. In Chapter 3, we generate entries in the order of tens of milliseconds per entry. Ma and Tsudik, for their publicly verifiable FssAgg

[9] https://github.com/trevp/axolotl/wiki, accessed 2015-01-29.

schemes, achieve entry generation (signing) in the order of milliseconds per entry [75] (using significantly older hardware than us). Marson and Poettering, with their seekable sequential key generators, generate *key material* in a few microseconds [76]. Similarly, Yavuz *et al.* [124] sign an entry in a few microseconds. For PillarBox, Bowers *et al.* [31] generate entries in the order of hundreds of microseconds per entry, specifying an average time for entry generation at 163 microseconds when storing syslog messages. Syslog messages are at most 1 KiB, so the average for Insynd of 142 microseconds at 7000 entries per second is comparable. Insynd therefore improves greatly on related work on transparency logging, and shows comparable performance to state-of-the-art secure logging systems. Note that our earlier work, presented in Chapter 3, Insynd, and PillarBox include the time to encrypt messages in addition to providing integrity protection.



(a) Entries per second in a 2^{20} Balloon.

(b) Goodput in a 2^{20} Balloon.

Figure 6.3 A benchmark on inserting entries. Figure 6.3a shows the number of entries that can be inserted per second in a 2^{20} Balloon for different message sizes. Figure 6.3b shows the corresponding goodput in MiB/s for the data in Figure 6.3a.

6.7 Conclusions

Insynd provides a number of privacy and security properties for one-way messaging. Insynd's design is based around concepts from authenticated data structures, forward-secure key generation from the secure logging area, and ongoing work on secure messaging protocols. Insynd is built around Balloon, a forward-secure append-only authenticated data structure presented in Chapter 5, inspired and built upon a history tree system by Crosby and Wallach [38]. For forward-secure key generation, Insynd borrows from FssAgg by Ma and Tsudik [75] and forward-secure SKGs originating

from Schneier and Kelsey [19, 104]. Finally, Insynd shares similarities with Pond, TextSecure, and OTR [30], by evolving key material and supporting "self-healing" in case of a time-limited compromise by a passive adversary. Insynd offers comparable performance for entry generation to state-of-the-art secure logging systems, like PillarBox [31]. The definitions used to evaluate Insynd originate from the scheme presented in Chapter 3 on privacy-preserving transparency logging focused solely around concepts from secure logging. By building upon and relating Insynd to authenticated data structures and secure messaging we open up for further improvements in the future, e.g., in terms of definitions that better capture security and privacy properties in the setting.

The use of user-specific FssAgg authenticators acts as a deterrent for an adversary to create snapshots that modify or delete entries sent prior to compromise. This is similar to the approach taken by CONIKS [78]. In future work, we would like to strengthen this deterrent by enabling users to create *publicly verifiable proofs of inconsistency* in case of inconsistencies between the FssAgg authenticator and the snapshots produced by the controller. This would enable users to publicly shame misbehaving controllers, and presumably act as an even stronger deterrent. We think that user-specific verifiability coupled with publicly verifiable proofs of inconsistency is a promising approach for enforcing proper gossiping of snapshots.

Chapter 7

Applications

> You can't defend. You can't prevent. The only thing you can do is detect and respond.
>
> *Bruce Schneier*

Chapter 3 described how transparency logging could be used in healthcare to give patients access to the access log of their electronic health records. This chapter presents two more applications of privacy-preserving transparency logging from the EU FP7 research project A4Cloud[1].

7.1 The Data Track

The Data Track is an end-user TET for data subjects with its roots in the EU research projects PRIME[2] [89] and PrimeLife[3] [51], and has been further enhanced in A4Cloud [11]. The Data Track is focused around *data disclosures*, enabling users to explore the data they have disclosed to service providers through different visualisations. Figure 7.1 shows one such visualisation of the A4Cloud Data Track called the trace view. The trace view is divided into three parts: attributes, the user, and services. The attributes, on the top of the view, represents data items that the user

[1] http://www.a4cloud.eu/, accessed 2015-04-22.
[2] https://www.prime-project.eu/, accessed 2015-04-22.
[3] http://www.primelife.eu/, accessed 2015-04-22.

has disclosed to one or more online services. In the middle is a picture representing the user. At the bottom of the trace view are online services that the user has directly or indirectly disclosed data to. By clicking on a service the trace view shows a trace (draws lines) between the service and the attributes the user has disclosed to the service. Similarly, clicking on an attribute shows all services that the user has disclosed the attribute to. Another visualisation of the Data Track is the timeline view, which shows data disclosures on a timeline. The idea is that by giving users different playful visualisations of their data disclosures they can explore their online digital footprints.

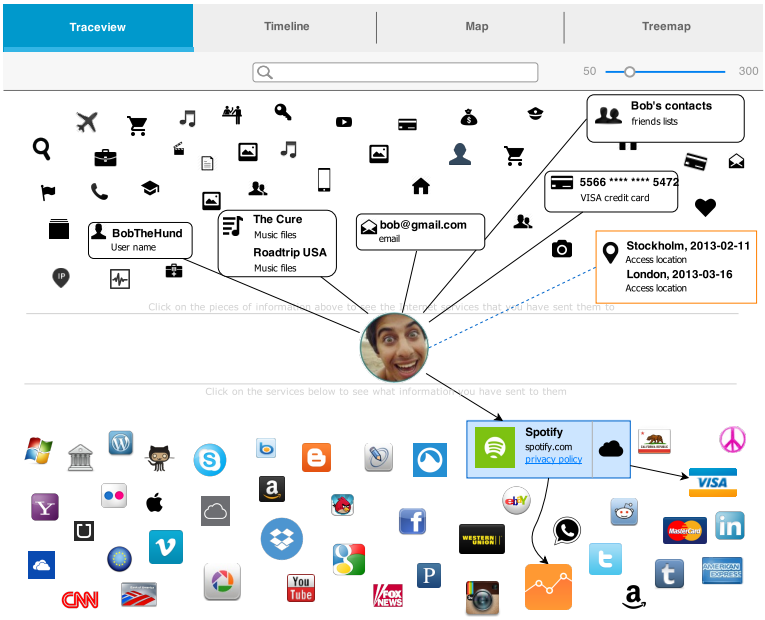


Figure 7.1 The Data Track trace view, visualising data disclosures of attributes to services.

In A4Cloud, the Data Track is the hub for a range of tools intended for data subjects of accountable cloud services. The data subject tools interact with other A4Cloud tools running at the cloud service provider (CSP). Figure 7.2 gives an overview of the relevant A4Cloud tool interactions for a CSP. This particular CSP is hosting the CardioMon service which is a web application for tracking health data. CaridoMon uses the accountable-PrimeLife policy language engine (A-PPLE) to manage its obligations with regard to the privacy policy of the service [14, 42]. A-PPLE attempts to assist the CSP with enforcing obligations like purpose binding,

data retention time, and notifications of incidents. When the CSP, either manually or with the help of other tools in the A4Cloud toolset (not shown in the figure, but one example is presented in Section 7.2), detects an incident the incident management tool (IMT) comes into play. IMT assists the CSP in dealing with incidents, potentially leading to the decision that the CSP is obliged to notify data subjects. If so, the IMT notifies A-PPLE and A-PPLE in turn uses a local service implementing the Insynd controller to notify data subjects. Each data subject is an Insynd user that queries an Insynd server for messages from the CSP. In case that a message is a notification of a potential privacy policy violation, the Data Track uses the plug-in for assessing policy violations (PAPV) to assess the severity of the violation. Based on severity, the Data Track displays the violation more or less prominently to the user. When the user wishes to deal with the notification, the Data Track invokes the remediation and redress tool (RRT). RRT provides users with different remediation and redress options based on the type of violation and privacy policy.

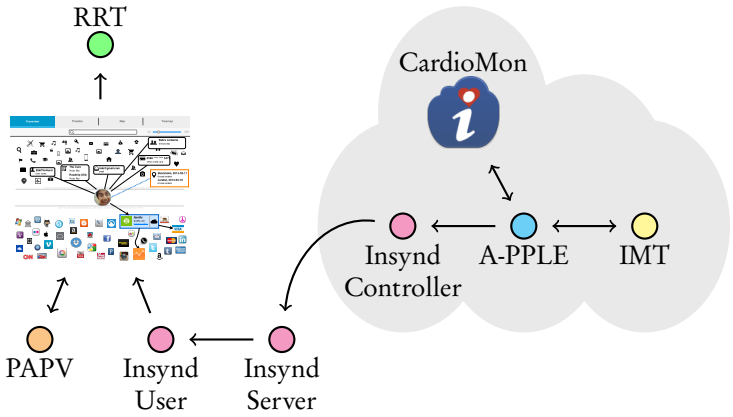


Figure 7.2 An overview of the relevant A4Cloud tool-set for the Data Track.

For data subjects, the A4Cloud tool-set uses Insynd for making data processing transparent and transporting notifications. Next, we describe these two applications in more detail and highlight the benefits of using Insynd.

7.1.1 Data Processing

A-PPLE is at the center of all data processing that occurs at the CSP. When CardioMon performs any processing on personal data it interacts with A-PPLE to read, write, or update personal data, thus A-PPLE acts as a middle layer between the application and a database of all personal data stored in the service (at least the data covered by the privacy policy). When sending a request to A-PPLE, CardioMon specifies the purpose of the action in the request. A-PPLE enforces that the request is in line with the agreed-upon privacy policy at data disclosure and triggers any obligations associated to the request. One possible obligation is to, on any request, log the request for the CSP or all relevant data subjects whose personal data is being requested, or both. When a request is logged for a data subject, A-PPLE uses Insynd.

When requests are logged for the data subject this enables the data subject to counter profile the CSP, as discussed in Section 1.2. The counter profiling can in part take the form of looking for policy violations beyond the aspects covered by the machine-readable privacy policy enforced by A-PPLE, such as related to the frequency of access to personal data potentially with respect to the purpose[4]. Another use case of this processing data is to derive data disclosures for the Data Track. When personal data is disclosed to the service, presumably something that happens frequently in the CardioMon service due to registering, e.g., hearth-rate over time using a medical device, these disclosures should also be made transparent to the data subject through the Data Track. Also, Insynd supports distributed settings. When the CSP discloses data downstream to another controller, that controller can continue making data processing transparent. In this case, the Data Track can reconstruct the data disclosure and make it transparent to the data subject thanks to Insynd.

The strong privacy properties of Insynd protects the user from leaking information to the CSP as part of receiving the logs, assuming the user takes necessary precautions when reconstructing, as discussed in Section 6.3.3. Furthermore, the CSP cannot after the fact change the logged messages describing data processing thanks to the deletion-detection forward integrity property of Insynd, and potentially the public verifiability, if the CSP uses an adequate gossiping mechanism for snapshots. Insynd also increases the utility of the data logged through it. If the data subject, e.g., would like to complain to a third party like a data protection agency (DPA) about any

[4]Consider the scenario where the CardioMon service needs a data subject's address for delivery of a medical device. Access for delivery purposes is presumably limited to the period of time before and during delivery. After delivery, access to an address for delivery purposes should presumably be infrequent although in line with the privacy policy. Frequent access may signal, e.g., that the CardioMon service has been compromised.

seemingly suspicious data processing Insynd enables the creation of publicly verifiable proofs of controller, message, user, and time. Finally, the fact that all messages sent through Insynd are periodically timestamped by a third party, enables the data subject to partially order multiple log trails generated using Insynd or any other source with strong proofs of time.

7.1.2 Notifications

As previously mentioned, A-PPLE sends notifications from the CSP to data subjects through Insynd. The use of Insynd over, e.g., email or push notification services provide a number of privacy and security benefits to data subjects, as discussed for making data processing transparent through Insynd. When the Data Track provides the notification potentially to the RRT (if an incident), the RRT gets a publicly verifiable proof binding the notification to the controller and a strong proof of when the notification was sent, relative to timestamps (granularity depends on the frequency of timestamps by the Insynd controller). What utility this has for the data subject ultimately depends (in A4Cloud) on the remediation and redress options provided by the RRT. One can imagine scenarios involving sharing the notification with a DPA or a consumer rights organisation. Another option is for data subjects to simply share the notification with the public to name and shame the CSP. Thanks to Insynd, anyone can verify that the controller sent the notification in question. Presumably, this acts as a deterrent for the CSP to misbehave.

At the services side, one benefit of Insynd is the support for distributed settings. As personal data is shared along a cloud supply chain, the use of Insynd ensures that even downstream controllers have the ability to notify data subjects. This may be extra relevant in the case of breach notifications, where timely breach notification is required for legal compliance with, e.g., the EU e-Privacy Directive 2002/58/EC, HIPAA, the proposed Personal Data Notification and Protection Act in the US, and the ongoing EU Data Protection Regulation. Insynd also enables the Insynd controller to create publicly verifiable proofs of controller, user, message, and time at the time of entry creation. This means that the CSP or other services can provide publicly verifiable *positive evidence* of providing timely breach notifications. Such evidence may serve as an account towards a DPA as part of a service fulfilling its legal obligations.

7.2 An Evidence Store for the Audit Agent System

The Audit Agent System (AAS) is an agent-based evidence collection system tailored for auditing complex systems and settings, such as cloud services. A central part of auditing is *evidence collection*. Centralised logging mechanisms, like security information and event management (SIEM) systems, are not enough to capture all relevant evidence in cloud settings. For example, stored data in databases, configuration files, and in cloud management systems can be of paramount importance and not necessarily available in SIEMs or any other centralised source. AAS therefore takes an *agent*-based approach[5], where custom agents can automate the *collection*, *storage* and *processing* of evidence, tailored to particular auditing tasks. In this section, we look at how to use Insynd as an evidence store for AAS, and what the potential gains are.

7.2.1 The Core Principles of Evidence

Mohay [82] describes the core principles of any evidence as admissibility, authenticity, completeness, reliability, and believability. These principles apply to common evidence as well as digital evidence, and therefore an evidence collection process has to take the principles into account. Next, we look at what Insynd has to offer for each principle.

Admissibility

The admissibility principle states that evidence must conform to certain legal rules, before it can be put before a jury. Admissibility of digital evidence is influenced by the transparency of the collection process and data protection regulation. Digital evidence can be any kind of data, like e-mail, social network messages, files, logs and so on. Insynd does not have any direct influence on the admissibility of the evidence stored in it.

Authenticity

The authenticity principle states that evidence must be linkable to the incident and may not be manipulated. Authenticity of digital evidence before court is closely related to the integrity requirement put on evidence records. Evidence may not be manipulated in any way and must be protected against any kind of tampering

[5] AAS is built on top of the Java agent development framework (JADE), http://jade.tilab.com, accessed 2015-04-22.

(willingly and accidentally). Insynd ensures that data cannot be tampered with once it is stored.

Completeness

The completeness principle states that evidence must be viewpoint agnostic and tell the whole story. Completeness is not directly ensured by Insynd, but rather needs to be ensured by the evidence collection process as a whole. Especially important are the definition of which evidence sources provide relevant evidence that need to be considered during the collection phase. Insynd can complement the evidence collection process by providing assurance of that all data stored in the evidence store are made available as evidence, and not cherry-picked.

Reliability

The reliability principle states that there cannot be any doubts about the evidence collection process and its correctness. Reliability is indirectly supported by integrating necessary mechanisms into the evidence collection process, such as Insynd.

Believability

The believability principle states that evidence must be understandable by a jury. Believability of the collected evidence is not influenced by implemented mechanisms, but rather by the interpretation and presentation by an expert in court. This is due to judges and juries usually being non-technical, which requires an abstracted presentation of evidence. Insynd does not influence the believability in that sense.

7.2.2 Evidence and Privacy

We considered four key privacy principles: confidentiality, data minimisation, purpose binding, and retention time. Protecting privacy in the process of evidence collection is utmost importance, since the collected data is likely to contain personal data.

Confidentiality

Confidentiality of data evolves around mechanisms for the protection from unwanted and unauthorized access. Typically, cryptographic concepts, such as encryption, are

used to ensure confidentiality of data. Confidentiality is a central property of Insynd by encrypting all data.

Data Minimisation

Data minimisation states that the collection of personal data should be minimised and limited to only what is strictly necessary. Collection agents are always configured for a specific audit task, which is limited in scope of what needs to be collected. Agents are never configured to arbitrarily collect data, but are always limited to a specific source (e.g., a server log) and data objects (e.g., a type of log entries). Insynd provides forward unlinkability of entries and client identifiers, which helps prevent several types of information leaks related to storing and accessing data.

Purpose Binding

Purpose binding of personal data entails that personal data should only be used for the purposes it was collected for. Neither Insynd nor our evidence process can directly influence the purpose for which collected data is used. Indirectly, the use of an evidence process like the one of AAS, incorporating secure evidence collection and storage, may serve to differentiate data collected for auditing purposes with other data collected, e.g., for marketing purposes.

Retention Time

Retention time is concerned with how long personal data may be stored and used before it needs to be deleted. These periods are usually defined by legal and business requirements. In cloud computing, the precise location of a data object is usually not directly available, i.e., the actual storage medium used to store a particular block of data is unknown, making data deletion hard. However, if data has been encrypted before storage, a reasonably safe way to ensure "deletion" is to discard the key material required for decryption. Insynd can be extended to support forward-secure clients, where key material to decrypt messages is discarded as messages are read.

7.2.3 An Evidence Collection Process and Insynd

Figure 7.3 shows an overview of the AAS evidence collection setting at a CSP. The AAS controller is a trusted system that controls AAS at a CSP. Each CSP has its own AAS controller. The CSP gives auditors access to the AAS controller, and from the

AAS controller auditors can create *auditing tasks*. An auditing task deploys one or more agents to the environment of the CSP to collect evidence. The agents store all collected evidence in the evidence store of the CSP. The evidence processor, another trusted system (also implemented as agents in JADE) query the evidence store for evidence and produces a *report*. The report is presented to the auditor as the result of the auditing task. Finally, through the AAS controller, the auditor can share the report with other CSPs and auditors.

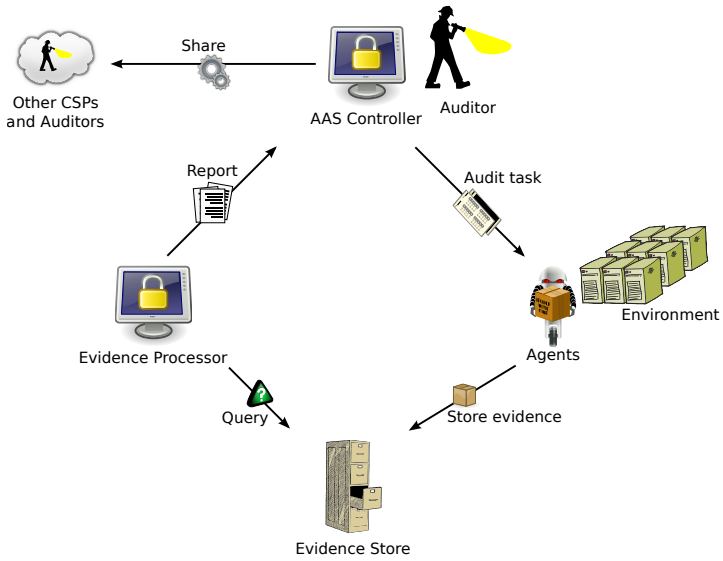


Figure 7.3 An overview of the AAS collection process at a cloud service provider (CSP).

Insynd fits into the process as follows. First, the evidence store is implemented as an Insynd server. The AAS controller, when deploying a collection agent, makes the agent into an Insynd controller while the corresponding evidence processor (also an agent) is made into an Insynd user. This way, the AAS controller isolates access to the evidence store for the evidence processor to the evidence of the relevant auditing task. Another benefit is that collection agents are forward secure and has no ability to modify or read any stored evidence collected prior to compromise of the agent, should it happen in the exposed environment of the CSP. Finally, when the evidence processor processes evidence from the evidence store it can verify that no tampering has taken place. When it creates the resulting report, it can include publicly verifiable proofs of controller, message and time of selected key evidence.

Assuming that the AAS controller certified the signing key of the collection agents, this creates strong additional proof of the reliability and authenticity of the evidence. Regarding completeness, the non-membership proofs of Insynd enable AAS to, if challenged ex-post an audit task, to show that all evidence (stored in entries) generated by a particular agent has been accounted for. Ensuring retention time of evidence in the evidence store is as simple as timely discarding Insynd user setup data on a per audit-task basis.

Chapter 8

Final Remarks

> I grew up with the understanding that the world I lived in was one where people enjoyed a sort of freedom to communicate with each other in privacy without it being monitored. Without it being measured or analyzed or sort of judged by these shadowy figures or systems any time they mention anything that travels across public lines.
>
> *Edward Snowden*

PETs are more relevant than ever. Information technology is entering every aspect of our lives and transforming society as we know it, for better or for worse. The Snowden revelations have brought privacy onto the center stage of public debate by revealing the systematic mass surveillance that the technological transformation has enabled. The need for transparency and accountability abound. Some defend the mass surveillance when performed by "them", but condemn the same actions by "others" as abuse of human rights. Hope for legal protection alone seems bleak. Technological means of protection, like PETs, may be our last resort for informational self-determination.

TETs promise to enhance transparency towards users, providing insights into how their privacy is affected by their actions through counter profiling. For TETs

that rely on information from the controllers of users' personal data, like the data provided through transparency logging, some trust in controllers is inevitable. This is surely not perfect, especially when looked at from the grim vantage point provided by the Snowden revelations, but the question is if it is good enough? Time will hopefully tell. One thing is for sure, that without transparency there is little hope for users to hold controllers accountable for their actions: you cannot hold an entity accountable for the unknown.

The subject of this dissertation is the construction of privacy-preserving transparency logging. Our work has contributed to three generations of cryptographic schemes for the setting. The first generation, as part of the work by Hedbom *et al.* [62], identified a key privacy property and was part of defining the privacy-preserving transparency logging setting. The second generation provided the first provably secure scheme with several definitions of security and privacy in the setting [95]. The third generation, built on top of an authenticated data structure customised for privacy-preserving transparency logging [93], addresses a number of limitations in the second generation with significantly improved performance and a focus on publicly verifiable consistency [94]. Furthermore, the third generation supports publicly verifiable proofs of controller, user, message, and time, increasing the utility of all data sent through the scheme.

We note several interesting areas for future work to further improve privacy for transparency logging, namely inconsistency proofs, trade-offs in reconstruction, and formalising user forward-security.

8.1 Inconsistency Proofs

The existence of a perfect gossiping mechanism for commitments (snapshots) on authenticated data structures underpins the security of, e.g., the History Tree system by Crosby and Wallach [38] (used as part of Balloon) and the closely related Certificate Transparency project by Laurie *et al.* [72]. For Insynd, we create an auxiliary mechanism for deletion-detection forward integrity, tied into Balloon. The mechanism can be privately verified by a user for his or her data as part of Balloon. This enables users to detect any inconsistency with regard to their entries in the Balloon, as indicated by the private mechanism, and as indicated by the response from membership queries in Balloon. This may deter the controller from ever creating inconsistent snapshots due to the risk of being detected. However, a user who detects an inconsistency cannot easily convince a third party of this fact in the current Insynd scheme, without at

least revealing its registration data and the contents of its last entry (due to the authenticator value being placed in it). Ideally, the user should be able to create a publicly verifiable proof of inconsistency with minimal impact on the user's privacy if he or she ever detects an inconsistency between its private verifiability mechanism and any snapshot. While this does nothing to increase the probability of detecting (which largely depends on the exact gossiping mechanism), it increases the potential impact of detection on the controller. In the end, this may act as a strong deterrent against incorrect gossiping, removing the need to assume a perfect gossiping mechanism.

8.2 Trade-Offs in Reconstruction

Our work has focused more on entry generation than entry reconstruction when it comes to privacy. Section 6.3.3 listed a number of approaches for privacy-preserving downloading of entries, but none has been investigated in detail. In addition to investigating the effectiveness of those different trade-offs in detail, another potential trade-off involves the use of Merkle aggregation [38]. Merkle aggregation enables extra data to be encoded into a Merkle tree, aggregating the data towards the root for each level. This could be used to aggregate the number of entries under each node in the hash treap as part of Balloon. With this information, users could download a verifiable subset of entries by having the server reveal parts of the hash treap in the "direction" of one or more entries of interest. Once a subset of entries of adequate size has been found, the server returns the result of membership queries for each entry in the subset.

8.3 Formalising User Forward-Security

Primarily as a consequence of scope, our work so far assumes that all users are trusted. This is naturally not always the case. A foundation for user forward-security is in place with the need of the (initial) authentication key to generate the nonce for decrypting entries (which could be continuously discarded), but not formalised or thoroughly analysed. Taking the compromise of users into account would also move us closer to the setting of closely related work on secure messaging schemes, which is an active area of research at the time of writing. As noted in Chapter 6, our approach already shares similarities, and benefiting from work from that area may simplify the construction of future privacy-preserving transparency logging schemes with stronger properties.

References

[1] R. Accorsi. Towards a secure logging mechanism for dynamic systems. In *In Proceedings of the 7th IT Security Symposium. São José dos Campos, Brazil. November 2005.*

[2] R. Accorsi. On the relationship of privacy and secure remote logging in dynamic systems. In *Security and Privacy in Dynamic Environments, Proceedings of the IFIP TC-11 21st International Information Security Conference (SEC 2006), 22–24 May 2006, Karlstad, Sweden*, pages 329–339, 2006.

[3] R. Accorsi. Log data as digital evidence: What secure logging protocols have to offer? In S. I. Ahamed, E. Bertino, C. K. Chang, V. Getov, L. Liu, H. Ming, and R. Subramanyan, editors, *COMPSAC (2)*, pages 398–403. IEEE Computer Society, 2009.

[4] R. Accorsi. Safe-keeping digital evidence with secure logging protocols: State of the art and challenges. In O. Goebel, R. Ehlert, S. Frings, D. Günther, H. Morgenstern, and D. Schadt, editors, *IMF*, pages 94–110. IEEE Computer Society, 2009.

[5] R. Accorsi. Bbox: A distributed secure log architecture. In J. Camenisch and C. Lambrinoudakis, editors, *Public Key Infrastructures, Services and Applications - 7th European Workshop, EuroPKI 2010, Athens, Greece, September 23-24, 2010. Revised Selected Papers*, volume 6711 of *Lecture Notes in Computer Science*, pages 109–124. Springer, 2010.

[6] R. Accorsi and A. Hohl. Delegating secure logging in pervasive computing systems. In *SPC*, pages 58–72, 2006.

[7] J. H. An. Authenticated encryption in the public-key setting: Security notions and analyses. *IACR Cryptology ePrint Archive*, 2001:79, 2001.

[8] A. Anagnostopoulos, M. T. Goodrich, and R. Tamassia. Persistent authenticated dictionaries and their applications. In G. I. Davida and Y. Frankel, editors, *ISC*, volume 2200 of *Lecture Notes in Computer Science*, pages 379–393. Springer, 2001.

[9] J. Angulo, S. Fischer-Hübner, T. Pulls, and U. König. HCI for policy display and administration. In J. Camenisch, S. Fischer-Hübner, and K. Rannenberg, editors, *Privacy and Identity Management for Life*, pages 261–277. Springer, 2011.

[10] J. Angulo, S. Fischer-Hübner, T. Pulls, and E. Wästlund. Towards usable privacy policy display & management - the primelife approach. In S. Furnell and N. L. Clarke, editors, *5th International Symposium on Human Aspects of Information Security and Assurance, HAISA 2011, London, UK, July 7-8, 2011. Proceedings*, pages 108–118. University of Plymouth, 2011.

[11] J. Angulo, S. Fischer-Hübner, T. Pulls, and E. Wästlund. Usable transparency with the data track: A tool for visualizing data disclosures. In *Proceedings of the 33rd Annual ACM Conference Extended Abstracts on Human Factors in Computing Systems*, CHI '15, pages 1803–1808, Seoul, Republic of Korea, 2015. ACM.

[12] J. Angulo, S. Fischer-Hübner, E. Wästlund, and T. Pulls. Towards usable privacy policy display & management for primelife. *Information Management & Computer Security*, 20(1):4–17, 2012.

[13] C. R. Aragon and R. Seidel. Randomized Search Trees. In *FOCS*, pages 540–545. IEEE Computer Society, 1989.

[14] M. Azraoui, K. Elkhiyaoui, M. Önen, K. Bernsmed, A. De Oliveira, and J. Sendor. A-ppl: An accountability policy language. In J. Garcia-Alfaro, J. Herrera-Joancomartí, E. Lupu, J. Posegga, A. Aldini, F. Martinelli, and N. Suri, editors, *Data Privacy Management, Autonomous Spontaneous Security, and Security Assurance*, volume 8872 of *Lecture Notes in Computer Science*, pages 319–326. Springer International Publishing, 2015.

[15] D. A. Basin, C. J. F. Cremers, T. H. Kim, A. Perrig, R. Sasse, and P. Szalachowski. ARPKI: attack resilient public-key infrastructure. In G. Ahn, M. Yung, and N. Li, editors, *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, November 3-7, 2014*, pages 382–393. ACM, 2014.

[16] M. Bellare, A. Boldyrevay, A. Desaiz, and D. Pointcheval. Key-Privacy in Public-Key Encryption. In C. Boyd, editor, *Advances in Cryptology – ASIACRYPT '01*, volume 2248 of *LNCS*, pages 566–582. Springer, 2001.

[17] M. Bellare, A. Desai, D. Pointcheval, and P. Rogaway. Relations Among Notions of Security for Public-Key Encryption Schemes. In H. Krawczyk, editor, *Advance in Cryptology – CRYPTO '98*, volume 1462 of *LNCS*, pages 26–45. Springer, 1998.

[18] M. Bellare and B. S. Yee. Forward Integrity For Secure Audit Logs. Technical report, 1997.

[19] M. Bellare and B. S. Yee. Forward-security in private-key cryptography. In M. Joye, editor, *Topics in Cryptology - CT-RSA 2003, The Cryptographers' Track at the RSA Conference 2003, San Francisco, CA, USA, April 13-17, 2003, Proceedings*, volume 2612 of *Lecture Notes in Computer Science*, pages 1–18. Springer, 2003.

[20] F. Bergadano, D. Cavagnino, P. D. Checco, P. A. Nesta, M. Miraglia, and P. L. Zaccone. Secure logging for irrefutable administration. *International Journal of Network Security*, 4(3):340–347, May 2007.

[21] D. J. Bernstein. The Poly1305-AES Message-Authentication Code. In *FSE*, volume 3557 of *LNCS*, pages 32–49. Springer, 2005.

[22] D. J. Bernstein. Curve25519: New Diffie-Hellman Speed Records. In M. Yung, Y. Dodis, A. Kiayias, and T. Malkin, editors, *Public Key Cryptography*, volume 3958 of *Lecture Notes in Computer Science*, pages 207–228. Springer, 2006.

[23] D. J. Bernstein. Extending the Salsa20 nonce. In *Symmetric Key Encryption Workshop*, 2011.

[24] D. J. Bernstein, N. Duif, T. Lange, P. Schwabe, and B.-Y. Yang. High-speed high-security signatures. *J. Cryptographic Engineering*, 2(2):77–89, 2012.

[25] D. J. Bernstein, T. Lange, and P. Schwabe. The Security Impact of a New Cryptographic Library. In A. Hevia and G. Neven, editors, *LATINCRYPT*, volume 7533 of *Lecture Notes in Computer Science*, pages 159–176. Springer, 2012.

[26] S. Berthold, S. Fischer-Hübner, L. Martucci, and T. Pulls. Crime and punishment in the cloud : Accountability, transparency, and privacy. DIMACS/BIC/A4Cloud/CSA International Workshop on Trustworthiness, Accountability and Forensics in the Cloud (TAFC), 2013.

[27] M. Blum, W. S. Evans, P. Gemmell, S. Kannan, and M. Naor. Checking the correctness of memories. *Algorithmica*, 12(2/3):225–244, 1994.

[28] D. Boneh and M. Franklin. Identity-based encryption from the weil pairing. In J. Kilian, editor, *Advances in Cryptology – CRYPTO 2001*, volume 2139 of *Lecture Notes in Computer Science*, pages 213–229. Springer, 2001.

[29] D. Boneh, B. Lynn, and H. Shacham. Short signatures from the weil pairing. *J. Cryptology*, 17(4):297–319, 2004.

[30] N. Borisov, I. Goldberg, and E. A. Brewer. Off-the-record communication, or, why not to use PGP. In V. Atluri, P. F. Syverson, and S. D. C. di Vimercati, editors, *WPES*, pages 77–84. ACM, 2004.

[31] K. D. Bowers, C. Hart, A. Juels, and N. Triandopoulos. PillarBox: Combating Next-Generation Malware with Fast Forward-Secure Logging. In *Research in Attacks, Intrusions and Defenses Symposium*, volume 8688 of *LNCS*, pages 46–67. Springer, 2014.

[32] A. Buldas, P. Laud, H. Lipmaa, and J. Willemson. Time-Stamping with Binary Linking Schemes. In *CRYPTO*, volume 1462 of *LNCS*, pages 486–501. Springer, 1998.

[33] J. Camenisch and E. Van Herreweghen. Design and implementation of the *idemix* anonymous credential system. In V. Atluri, editor, *Proceedings of the 9th ACM Conference on Computer and Communications Security, CCS 2002, Washington, DC, USA, November 18-22, 2002*, pages 21–30. ACM, 2002.

[34] P. Chodowiec and K. Gaj. Very compact FPGA implementation of the AES algorithm. In C. D. Walter, Ç. K. Koç, and C. Paar, editors, *Cryptographic Hardware and Embedded Systems - CHES 2003, 5th International Workshop, Cologne, Germany, September 8-10, 2003, Proceedings*, volume 2779 of *Lecture Notes in Computer Science*, pages 319–333. Springer, 2003.

[35] C. N. Chong, Z. Peng, and P. H. Hartel. Secure audit logging with tamper-resistant hardware. In *18th IFIP International Information Security Conference (IFIP SEC), volume 250 of IFIP Conference Proceedings*, pages 73–84. Kluwer Academic Publishers, 2002.

[36] B. Chor, O. Goldreich, E. Kushilevitz, and M. Sudan. Private information retrieval. In *36th Annual Symposium on Foundations of Computer Science, Milwaukee, Wisconsin, 23-25 October 1995*, pages 41–50. IEEE Computer Society, 1995.

[37] S. A. Crosby. *Efficient tamper-evident data structures for untrusted servers*. PhD thesis, Rice University, Houston, TX, USA, 2010.

[38] S. A. Crosby and D. S. Wallach. Efficient Data Structures For Tamper-Evident Logging. In *USENIX Security Symposium*, pages 317–334. USENIX Association, 2009.

[39] S. A. Crosby and D. S. Wallach. Super-Efficient Aggregating History-Independent Persistent Authenticated Dictionaries. In M. Backes and P. Ning, editors, *ESORICS*, volume 5789 of *Lecture Notes in Computer Science*, pages 671–688. Springer, 2009.

[40] S. A. Crosby and D. S. Wallach. Authenticated dictionaries: Real-world costs and trade-offs. *ACM Trans. Inf. Syst. Secur.*, 14(2):17, 2011.

[41] I. Damgård. A "proof-reading" of some issues in cryptography. In *Automata, Languages and Programming*, pages 2–11. Springer, 2007.

[42] A. de Oliveira, J. Sendor, V. o Thành Phúc, E. Vlachos, M. Azraoui, K. Elkhiyaoui, M. Önen, W. Benghabrit, J.-C. Royer, M. D'Errico, M. G. Jaatun, and I. A. Tøndel. D-43.2: Prototype for accountability enforcement tools and services. Technical Report D:D-43.2, Accountability for Cloud and Future Internet Services - A4Cloud Project, 2015.

[43] C. Devet and I. Goldberg. The Best of Both Worlds: Combining Information-Theoretic and Computational PIR for Communication Efficiency. In *PETS*, volume 8555 of *LNCS*, pages 63–82. Springer, 2014.

[44] C. Diaz, O. Tene, and S. Gurses. Hero or villain: The data controller in privacy law and technologies. *Ohio St. LJ*, 74:923, 2013.

[45] R. Dingledine, N. Mathewson, and P. F. Syverson. Tor: The Second-Generation Onion Router. In *USENIX Security Symposium*, pages 303–320. USENIX, 2004.

[46] G. Dodig-Crnkovic. Scientific methods in computer science. In *Conference for the Promotion of Research in IT at New Universities and at University Colleges in Sweden*, April 2002.

[47] C. Dwork. Differential privacy. In M. Bugliesi, B. Preneel, V. Sassone, and I. Wegener, editors, *Automata, Languages and Programming, 33rd International Colloquium, ICALP 2006, Venice, Italy, July 10-14, 2006, Proceedings, Part II*, volume 4052 of *Lecture Notes in Computer Science*, pages 1–12. Springer, 2006.

[48] C. Fernandez-Gago, V. Tountopoulos, S. Fischer-Hübner, R. Alnemr, D. Nuñez, J. Angulo, T. Pulls, and T. Koulouris. Tools for cloud accountability: A4cloud tutorial. in: Camenisch et al.: Privacy and Identity Management for the Future Internet in the Age of Globalisation – Proceedings of the IFIP Summer School 2014. IFIP AICT 457, Springer, 2015.

[49] FIDIS WP7. D 7.12: Behavioural Biometric Profiling and Transparency Enhancing Tools. Future of Identity in the Information Society, http://www.fidis.net/resources/deliverables/profiling/, March 2009.

[50] S. Fischer-Hübner, J. Angulo, and T. Pulls. How can cloud users be supported in deciding on, tracking and controlling how their data are used? In M. Hansen, J. Hoepman, R. E. Leenes, and D. Whitehouse, editors, *Privacy and Identity Management for Emerging Services and Technologies - 8th IFIP WG 9.2, 9.5, 9.6/11.7, 11.4, 11.6 International Summer School, Nijmegen, The Netherlands, June 17-21, 2013, Revised Selected Papers*, volume 421 of *IFIP Advances in Information and Communication Technology*, pages 77–92. Springer, 2013.

[51] S. Fischer-Hübner, H. Hedbom, and E. Wästlund. Trust and assurance HCI. In J. Camenisch, S. Fischer-Hübner, and K. Rannenberg, editors, *Privacy and Identity Management for Life*, pages 245–260. Springer, 2011.

[52] A. Futoransky and E. Kargieman. VCR y PEO, dos protocolos criptogr'aficos simples. In *Jornadas Argentinas de Inform'atica e Investigaci'on Operativa*, July 1995.

[53] A. Futoransky and E. Kargieman. PEO revisited. In *DISC 98 – Di'a Internacional de la Seguridad en C'omputo. Mexico D.F., Mexico*, 1998.

[54] P. Godfrey-Smith. *Theory and Reality: An Introduction to the Philosophy of Science*. Science and Its Conceptual Foundations. University of Chicago Press, 2003.

[55] I. Goldberg, D. Wagner, and E. Brewer. Privacy-enhancing technologies for the internet. Technical report, DTIC Document, 1997.

[56] O. Goldreich. On post-modern cryptography. *IACR Cryptology ePrint Archive*, 2006:461, 2006.

[57] S. Goldwasser and S. Micali. Probabilistic encryption. *Journal of computer and system sciences*, 28(2):270–299, 1984.

[58] S. Goldwasser, S. Micali, and R. L. Rivest. A Digital Signature Scheme Secure Against Adaptive Chosen-Message Attacks. *SIAM Journal on Computing*, 17:281–308, 1988.

[59] H. Hedbom. A survey on transparency tools for enhancing privacy. In V. Matyás, S. Fischer-Hübner, D. Cvrcek, and P. Svenda, editors, *The Future of Identity in the Information Society - 4th IFIP WG 9.2, 9.6/11.6, 11.7/FIDIS International Summer School, Brno, Czech Republic, September 1-7, 2008, Revised Selected Papers*, volume 298 of *IFIP Advances in Information and Communication Technology*, pages 67–82. Springer, 2008.

[60] H. Hedbom and T. Pulls. Unlinking database entries: Implementation issues in privacy preserving secure logging. In *2010 2nd International Workshop on Security and Communication Networks (IWSCN)*, pages 1–7. IEEE, 2010.

[61] H. Hedbom, T. Pulls, and M. Hansen. Transparency tools. In J. Camenisch, S. Fischer-Hübner, and K. Rannenberg, editors, *Privacy and Identity Management for Life*, pages 135–143. Springer, 2011.

[62] H. Hedbom, T. Pulls, P. Hjärtquist, and A. Lavén. Adding secure transparency logging to the PRIME Core. In M. Bezzi, P. Duquenoy, S. Fischer-Hübner, M. Hansen, and G. Zhang, editors, *Privacy and Identity Management for Life*, volume 320 of *IFIP Advances in Information and Communication Technology*, pages 299–314. Springer Boston, 2010.

[63] J. Herranz, D. Hofheinz, and E. Kiltz. KEM/DEM: Necessary and Sufficient Conditions for Secure Hybrid Encryption. Cryptology ePrint Archive, Report 2006/265, 2006.

[64] J. E. Holt. Logcrypt: Forward Security and Public Verification for Secure Audit Logs. In R. Buyya, T. Ma, R. Safavi-Naini, C. Steketee, and W. Susilo, editors, *ACSW Frontiers '06*, volume 54 of *CRPIT*, pages 203–211. Australian Computer Society, 2006.

[65] ISO/IEC. ISO/IEC 11889-1–4; Information technology – Trusted Platform Module. http://www.iso.org/iso/catalogue_detail.htm?csnumber=50970, 2009.

[66] T. H. Kim, L. Huang, A. Perrig, C. Jackson, and V. D. Gligor. Accountable key infrastructure (AKI): a proposal for a public-key validation infrastructure. In D. Schwabe, V. A. F. Almeida, H. Glaser, R. A. Baeza-Yates, and S. B. Moon, editors, *22nd International World Wide Web Conference, WWW '13, Rio de Janeiro, Brazil, May 13-17, 2013*, pages 679–690. International World Wide Web Conferences Steering Committee / ACM, 2013.

[67] R. K. L. Ko, P. Jagadpramana, M. Mowbray, S. Pearson, M. Kirchberg, Q. Liang, and B.-S. Lee. TrustCloud: A Framework for Accountability and Trust in Cloud Computing. In *SERVICES*, pages 584–588. IEEE Computer Society, 2011.

[68] N. Koblitz and A. Menezes. Another look at "provable security". *IACR Cryptology ePrint Archive*, 2004:152, 2004.

[69] N. Koblitz and A. Menezes. Another look at "provable security". II. *IACR Cryptology ePrint Archive*, 2006:229, 2006.

[70] D. Lathouwers, K. Simoens, and K. Wouters. Process logging and visualisation across governmental institutions. K.U.Leuven/COSIC internal report, IBBT/Index Deliv 6.1, K.U.Leuven/IBBT, 2007.

[71] B. Laurie and E. Kasper. Revocation transparency. September 2012. http://www.links.org/files/RevocationTransparency.pdf.

[72] B. Laurie, A. Langley, and E. Kasper. Certificate Transparency. RFC 6962, June 2013.

[73] D. Ma. Practical forward secure sequential aggregate signatures. In M. Abe and V. D. Gligor, editors, *ASIACCS*, pages 341–352. ACM, 2008.

[74] D. Ma and G. Tsudik. Extended abstract: Forward-secure sequential aggregate authentication. In *IEEE Symposium on Security and Privacy*, pages 86–91. IEEE Computer Society, 2007.

[75] D. Ma and G. Tsudik. A new approach to secure logging. *TOS*, 5(1), 2009.

[76] G. A. Marson and B. Poettering. Even More Practical Secure Logging: Tree-Based Seekable Sequential Key Generators. In *ESORICS*, volume 8713 of *LNCS*, pages 37–54. Springer, 2014.

[77] C. U. Martel, G. Nuckolls, P. T. Devanbu, M. Gertz, A. Kwong, and S. G. Stubblebine. A General Model for Authenticated Data Structures. *Algorithmica*, 39(1):21–41, 2004.

[78] M. S. Melara, A. Blankstein, J. Bonneau, M. J. Freedman, and E. W. Felten. CONIKS: A privacy-preserving consistent key service for secure end-to-end communication. Cryptology ePrint Archive, Report 2014/1004, 2014.

[79] R. C. Merkle. A Digital Signature Based on a Conventional Encryption Function. In C. Pomerance, editor, *CRYPTO*, volume 293 of *Lecture Notes in Computer Science*, pages 369–378. Springer, 1987.

[80] R. C. Merkle. A Certified Digital Signature. In G. Brassard, editor, *CRYPTO*, volume 435 of *Lecture Notes in Computer Science*, pages 218–238. Springer, 1989.

[81] A. Miller, M. Hicks, J. Katz, and E. Shi. Authenticated data structures, generically. In S. Jagannathan and P. Sewell, editors, *POPL*, pages 411–424. ACM, 2014.

[82] G. M. Mohay. *Computer and intrusion forensics*. Artech House, 2003.

[83] A. Moradi, A. Barenghi, T. Kasper, and C. Paar. On the vulnerability of FPGA bitstream encryption against power analysis attacks: extracting keys from xilinx virtex-ii fpgas. In Y. Chen, G. Danezis, and V. Shmatikov, editors, *Proceedings of the 18th ACM Conference on Computer and Communications Security, CCS 2011, Chicago, Illinois, USA, October 17-21, 2011*, pages 111–124. ACM, 2011.

[84] K. Nissim and M. Naor. Certificate revocation and certificate update. In A. D. Rubin, editor, *Proceedings of the 7th USENIX Security Symposium, San Antonio, TX, USA, January 26-29, 1998*. USENIX Association, 1998.

[85] NIST. FIPS 186-3: Digital Signature Standard (DSS), 2009.

[86] C. Papamanthou, R. Tamassia, and N. Triandopoulos. Optimal verification of operations on dynamic sets. In *CRYPTO*, volume 6841 of *LNCS*, pages 91–110. Springer, 2011.

[87] S. Y. Park and A. K. Bera. Maximum Entropy Autoregressive Conditional Heteroskedasticity Model. *Journal of Econometrics*, 150(2):219–230, June 2009.

[88] R. Peeters, T. Pulls, and K. Wouters. Enhancing transparency with distributed privacy-preserving logging. In *ISSE 2013 Securing Electronic Business Processes*, pages 61–71. Springer, 2013.

[89] J. S. Pettersson, S. Fischer-Hübner, and M. Bergmann. Outlining "data track": Privacy-friendly data maintenance for end-users. In *Advances in Information Systems Development*, pages 215–226. Springer, 2007.

[90] G. Pólya. *How to solve it: a new aspect of mathematical method*. Science study series. Doubleday & Company, Inc, 1957.

[91] T. Pulls. (more) side channels in cloud storage - linking data to users. In J. Camenisch, B. Crispo, S. Fischer-Hübner, R. Leenes, and G. Russello, editors, *Privacy and Identity Management for Life - 7th IFIP WG 9.2, 9.6/11.7, 11.4, 11.6/PrimeLife International Summer School, Trento, Italy, September 5-9, 2011, Revised Selected Papers*, volume 375 of *IFIP Advances in Information and Communication Technology*, pages 102–115. Springer, 2011.

[92] T. Pulls. Privacy-friendly cloud storage for the data track - an educational transparency tool. In A. Jøsang and B. Carlsson, editors, *Secure IT Systems - 17th Nordic Conference, NordSec 2012, Karlskrona, Sweden, October 31 - November 2, 2012. Proceedings*, volume 7617 of *Lecture Notes in Computer Science*, pages 231–246. Springer, 2012.

[93] T. Pulls and R. Peeters. Balloon: A forward-secure append-only persistent authenticated data structure. Cryptology ePrint Archive, Report 2015/007, 2015.

[94] T. Pulls and R. Peeters. Insynd: Secure one-way messaging through Balloons. Cryptology ePrint Archive, Report 2015/150, 2015.

[95] T. Pulls, R. Peeters, and K. Wouters. Distributed privacy-preserving transparency logging. In *Proceedings of the 12th annual ACM Workshop on Privacy in the Electronic Society, WPES 2013, Berlin, Germany, November 4, 2013*, pages 83–94. ACM, 2013.

[96] T. Pulls and D. Slamanig. On the feasibility of (practical) commercial anonymous cloud storage. *Transactions on Data Privacy*, to appear.

[97] T. Pulls, K. Wouters, J. Vliegen, and C. Grahn. Distributed Privacy-Preserving Log Trails. Technical Report 2012:24, Karlstad University, Department of Computer Science, 2012.

[98] J. Roberts. No one is perfect: The limits of transparency and an ethic for 'intelligent' accountability. *Accounting, Organizations and Society*, 34(8):957–970, 2009.

[99] P. Rogaway. Practice-oriented provable security and the social construction of cryptography. *Unpublished essay*, 2009.

[100] T. Ruebsamen, T. Pulls, and C. Reich. Secure evidence collection and storage for cloud accountability audits. CLOSER 2015 - Proceedings of the 5th International Conference on Cloud Computing and Services Science, Lisbon, Portugal, May 20-22, 2015.

[101] M. D. Ryan. Enhanced Certificate Transparency and End-to-End Encrypted Mail. In *NDSS*. The Internet Society, 2014.

[102] D. Rykx and J. Thielen. Evaluatie van nieuwe hashfunctie kandidaten op fpga, 2011.

[103] S. Sackmann, J. Strüker, and R. Accorsi. Personalization in Privacy-Aware Highly Dynamic Systems. *Communications of the ACM*, 49(9):32–38, 2006.

[104] B. Schneier and J. Kelsey. Cryptographic Support for Secure Logs on Untrusted Machines. In *USENIX Security Symposium '98*, pages 53–62. USENIX, 1998.

[105] B. Schneier and J. Kelsey. Event auditing system, November 1999. U.S. Patent 5978475.

[106] B. Schneier and J. Kelsey. Secure audit logs to support computer forensics. *ACM Trans. Inf. Syst. Secur.*, 2(2):159–176, 1999.

[107] V. Shoup. Sequences of Games: a Tool for Taming Complexity in Security Proofs. Cryptology ePrint Archive, Report 2004/332, 2004.

[108] V. Shoup. ISO 18033-2:2006 Information technology – Security techniques – Encryption algorithms – Part 2: Asymmetric ciphers, May 2006.

[109] N. P. Smart. The Exact Security of ECIES in the Generic Group Model. In B. Honary, editor, *Cryptography and Coding*, volume 2260 of *LNCS*, pages 73–84. Springer, 2001.

[110] V. Stathopoulos, P. Kotzanikolaou, and E. Magkos. A framework for secure and verifiable logging in public communication networks. In J. López, editor, *CRITIS*, volume 4347 of *Lecture Notes in Computer Science*, pages 273–284. Springer, 2006.

[111] V. Stathopoulos, P. Kotzanikolaou, and E. Magkos. Secure log management for privacy assurance in electronic communications. *Computers & Security*, 27(7–8):298–308, 2008.

[112] R. Tamassia. Authenticated data structures. In *Algorithms - ESA 2003*, volume 2832 of *LNCS*, pages 2–5. Springer, 2003.

[113] A. M. Turing. On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London Mathematical Society*, 42:230–265, July 1936.

[114] United Nations Department of Economic and Social Affairs. UN e-Government Survey 2012. E-Government for the People. Technical report, 2012.

[115] G. Van Blarkom, J. Borking, and J. Olk. Handbook of privacy and privacy-enhancing technologies. *Privacy Incorporated Software Agent (PISA) Consortium, The Hague*, 2003.

[116] J. Vliegen. *Partial and dynamic FPGA reconfiguration for security applications*. PhD thesis, KU Leuven, 2014. Nele Mentens and Ingrid Verbauwhede (promotors).

[117] J. Vliegen, N. Mentens, J. Genoe, A. Braeken, S. Kubera, A. Touhafi, and I. Verbauwhede. A compact fpga-based architecture for elliptic curve cryptography over prime fields. In F. Charot, F. Hannig, J. Teich, and C. Wolinski, editors, *21st IEEE International Conference on Application-specific Systems Architectures and Processors, ASAP 2010, Rennes, France, 7-9 July 2010*, pages 313–316. IEEE, 2010.

[118] J. Vliegen, K. Wouters, C. Grahn, and T. Pulls. Hardware strengthening a distributed logging scheme. In *15th Euromicro Conference on Digital System Design, DSD 2012, Cesme, Izmir, Turkey, September 5-8, 2012*, pages 171–176. IEEE Computer Society, 2012.

[119] B. R. Waters, D. Balfanz, G. Durfee, and D. K. Smetters. Building an encrypted and searchable audit log. In *Proceedings of the 11th Network and Distributed System Security Symposium, NDSS 2004, San Diego, California, USA*, pages 205–214, 2004.

[120] P. Winter, T. Pulls, and J. Fuß. Scramblesuit: a polymorphic network protocol to circumvent censorship. In A. Sadeghi and S. Foresti, editors, *Proceedings of the 12th annual ACM Workshop on Privacy in the Electronic Society, WPES 2013, Berlin, Germany, November 4, 2013*, pages 213–224. ACM, 2013.

[121] K. Wold and C. H. Tan. Analysis and enhancement of random number generator in FPGA based on oscillator rings. In *ReConFig'08: 2008 International Conference on Reconfigurable Computing and FPGAs, 3-5 December 2008, Cancun, Mexico, Proceedings*, pages 385–390. IEEE Computer Society, 2008.

[122] K. Wouters, K. Simoens, D. Lathouwers, and B. Preneel. Secure and Privacy-Friendly Logging for eGovernment Services. In *ARES*, pages 1091–1096. IEEE Computer Society, 2008.

[123] A. A. Yavuz and P. Ning. BAF: An Efficient Publicly Verifiable Secure Audit Logging Scheme for Distributed Systems. In *ACSAC*, pages 219–228. IEEE Computer Society, 2009.

[124] A. A. Yavuz, P. Ning, and M. K. Reiter. BAF and FI-BAF: efficient and publicly verifiable cryptographic schemes for secure logging in resource-constrained systems. *ACM Trans. Inf. Syst. Secur.*, 15(2):9, 2012.

[125] J. Yu, V. Cheval, and M. Ryan. DTKI: a new formalized PKI with no trusted parties. *CoRR*, abs/1408.1023, 2014.

Appendix A

List of Publications

This appendix contains a list of all papers, technical reports, and book chapters I have contributed to over the course of my PhD studies.

[9] J. Angulo, S. Fischer-Hübner, T. Pulls, and U. König. HCI for policy display and administration. In J. Camenisch, S. Fischer-Hübner, and K. Rannenberg, editors, *Privacy and Identity Management for Life*, pages 261–277. Springer, 2011

[10] J. Angulo, S. Fischer-Hübner, T. Pulls, and E. Wästlund. Towards usable privacy policy display & management - the primelife approach. In S. Furnell and N. L. Clarke, editors, *5th International Symposium on Human Aspects of Information Security and Assurance, HAISA 2011, London, UK, July 7-8, 2011. Proceedings*, pages 108–118. University of Plymouth, 2011

[11] J. Angulo, S. Fischer-Hübner, T. Pulls, and E. Wästlund. Usable transparency with the data track: A tool for visualizing data disclosures. In *Proceedings of the 33rd Annual ACM Conference Extended Abstracts on Human Factors in Computing Systems*, CHI '15, pages 1803–1808, Seoul, Republic of Korea, 2015. ACM

[12] J. Angulo, S. Fischer-Hübner, E. Wästlund, and T. Pulls. Towards usable privacy policy display & management for primelife. *Information Management & Computer Security*, 20(1):4–17, 2012

[26] S. Berthold, S. Fischer-Hübner, L. Martucci, and T. Pulls. Crime and punishment in the cloud : Accountability, transparency, and privacy. DIMACS/BIC/A4Cloud/CSA International Workshop on Trustworthiness, Accountability and Forensics in the Cloud (TAFC), 2013

[48] C. Fernandez-Gago, V. Tountopoulos, S. Fischer-Hübner, R. Alnemr, D. Nuñez, J. Angulo, T. Pulls, and T. Koulouris. Tools for cloud accountability: A4cloud tutorial. in: Camenisch et al.: Privacy and Identity Management for the Future Internet in the Age of Globalisation – Proceedings of the IFIP Summer School 2014. IFIP AICT 457, Springer, 2015

[50] S. Fischer-Hübner, J. Angulo, and T. Pulls. How can cloud users be supported in deciding on, tracking and controlling how their data are used? In M. Hansen, J. Hoepman, R. E. Leenes, and D. Whitehouse, editors, *Privacy and Identity Management for Emerging Services and Technologies - 8th IFIP WG 9.2, 9.5, 9.6/11.7, 11.4, 11.6 International Summer School, Nijmegen, The Netherlands, June 17-21, 2013, Revised Selected Papers*, volume 421 of *IFIP Advances in Information and Communication Technology*, pages 77–92. Springer, 2013

[60] H. Hedbom and T. Pulls. Unlinking database entries: Implementation issues in privacy preserving secure logging. In *2010 2nd International Workshop on Security and Communication Networks (IWSCN)*, pages 1–7. IEEE, 2010

[61] H. Hedbom, T. Pulls, and M. Hansen. Transparency tools. In J. Camenisch, S. Fischer-Hübner, and K. Rannenberg, editors, *Privacy and Identity Management for Life*, pages 135–143. Springer, 2011

[62] H. Hedbom, T. Pulls, P. Hjärtquist, and A. Lavén. Adding secure transparency logging to the PRIME Core. In M. Bezzi, P. Duquenoy, S. Fischer-Hübner, M. Hansen, and G. Zhang, editors, *Privacy and Identity Management for Life*, volume 320 of *IFIP Advances in Information and Communication Technology*, pages 299–314. Springer Boston, 2010

[88] R. Peeters, T. Pulls, and K. Wouters. Enhancing transparency with distributed privacy-preserving logging. In *ISSE 2013 Securing Electronic Business Processes*, pages 61–71. Springer, 2013

[91] T. Pulls. (more) side channels in cloud storage - linking data to users. In J. Camenisch, B. Crispo, S. Fischer-Hübner, R. Leenes, and G. Russello, editors, *Privacy and Identity Management for Life - 7th IFIP WG 9.2, 9.6/11.7, 11.4, 11.6/PrimeLife International Summer School, Trento, Italy, September 5-9, 2011, Revised Selected Papers*, volume 375 of *IFIP Advances in Information and Communication Technology*, pages 102–115. Springer, 2011

[92] T. Pulls. Privacy-friendly cloud storage for the data track - an educational transparency tool. In A. Jøsang and B. Carlsson, editors, *Secure IT Systems - 17th Nordic Conference, NordSec 2012, Karlskrona, Sweden, October 31 - November 2, 2012. Proceedings*, volume 7617 of *Lecture Notes in Computer Science*, pages 231–246. Springer, 2012

[93] T. Pulls and R. Peeters. Balloon: A forward-secure append-only persistent authenticated data structure. Cryptology ePrint Archive, Report 2015/007, 2015

[94] T. Pulls and R. Peeters. Insynd: Secure one-way messaging through Balloons. Cryptology ePrint Archive, Report 2015/150, 2015

[95] T. Pulls, R. Peeters, and K. Wouters. Distributed privacy-preserving transparency logging. In *Proceedings of the 12th annual ACM Workshop on Privacy in the Electronic Society, WPES 2013, Berlin, Germany, November 4, 2013*, pages 83–94. ACM, 2013

[96] T. Pulls and D. Slamanig. On the feasibility of (practical) commercial anonymous cloud storage. *Transactions on Data Privacy*, to appear

[97] T. Pulls, K. Wouters, J. Vliegen, and C. Grahn. Distributed Privacy-Preserving Log Trails. Technical Report 2012:24, Karlstad University, Department of Computer Science, 2012

[100] T. Ruebsamen, T. Pulls, and C. Reich. Secure evidence collection and storage for cloud accountability audits. CLOSER 2015 - Proceedings of the 5th International Conference on Cloud Computing and Services Science, Lisbon, Portugal, May 20-22, 2015

[118] J. Vliegen, K. Wouters, C. Grahn, and T. Pulls. Hardware strengthening a distributed logging scheme. In *15th Euromicro Conference on Digital System Design, DSD 2012, Cesme, Izmir, Turkey, September 5-8, 2012*, pages 171–176. IEEE Computer Society, 2012

[120] P. Winter, T. Pulls, and J. Fuß. Scramblesuit: a polymorphic network protocol to circumvent censorship. In A. Sadeghi and S. Foresti, editors, *Proceedings of the 12th annual ACM Workshop on Privacy in the Electronic Society, WPES 2013, Berlin, Germany, November 4, 2013*, pages 213–224. ACM, 2013

Preserving Privacy in Transparency Logging

The subject of this dissertation is the construction of privacy-enhancing technologies (PETs) for transparency logging, a technology at the intersection of privacy, transparency, and accountability. Transparency logging facilitates the transportation of data from service providers to users of services and is therefore a key enabler for ex-post transparency-enhancing tools (TETs). Ex-post transparency provides information to users about how their personal data have been processed by service providers, and is a prerequisite for accountability: you cannot hold a service provider accountable for what is unknown.

We present three generations of PETs for transparency logging to which we contributed. We start with early work that defined the setting as a foundation and build upon it to increase both the privacy protections and the utility of the data sent through transparency logging. Our contributions include the first provably secure privacy-preserving transparency logging scheme and a forward-secure append-only persistent authenticated data structure tailored to the transparency logging setting. Applications of our work range from notifications and deriving data disclosures for the Data Track (an ex-post TET) to secure evidence storage.

ISBN 978-91-7063-644-8

ISSN 1403-8099

DISSERTATION | Karlstad University Studies | 2015:28