Security for the cloud

Mario Cornejo-Ramirez

THÈSE DE DOCTORAT

de l'Université de recherche Paris Sciences et Lettres
PSL Research University

Préparée à l'École normale supérieure

Security for the Cloud

Ecole doctorale n°386

SCIENCES MATHÉMATIQUES DE PARIS CENTRE

Spécialité INFORMATIQUE

COMPOSITION DU JURY :

M. FERREIRA ABDALLA Michel
École normale supérieure
Directeur de thése

M. GOUBIN Louis
University of Versailles-St-Quentin-en-Yvelines
Rapporteur

M. LAGUILLAUMIE Fabien
Université de Lyon 1
Rapporteur

Mme. CHEVALIER Céline
Université Paris 2
Membre du jury

M. FOUQUE Pierre-Alain
Université Rennes 1
Membre du jury

M. NEVEN Gregory
IBM Research - Zurich,
Membre du jury

M. POINTCHEVAL David
École normale supérieure
Membre du jury

Soutenue par Mario
CORNEJO-RAMIREZ
le 17 novembre 2016

Dirigée par
Michel FERREIRA ABDALLA

ENS
ÉCOLE NORMALE SUPÉRIEURE

PSL
RESEARCH UNIVERSITY PARIS

Acknowledgments

First I would like to say thanks to my advisor Michel Abdalla. Michel has not just been a great advisor, but also a dear friend. Thanks for inviting me to do an internship during my masters and to participate in what would be the beginning of this adventure, for being always open to discussion, for your advice and for your help anytime I needed it, especially at the beginning. And foremost, thanks for convincing me to come to Paris. My deepest thanks to my David Pointcheval. Thanks for your enthusiasm, for being there every day I needed to discuss, for your moral support, and for replying to my emails in the very (very) late night.

I am also very grateful to my two reviewers Louis Goubin and Fabien Laguillaumie. Thanks for taking the time to review my thesis and improve it with your remarks and comments. I am also thankful to the other committee members: Michel Abdalla, Céline Chevalier, Pierre-Alain Fouque, Gregory Neven and David Pointcheval.

I would like to individually express my gratitude to all my co-authors: Michel Abdalla, Raphael Bost, Pierre-Alain Fouque, Anca Nitulescu, David Pointcheval and Sylvain Ruhault. It was very rewarding working with all of you.

I also would like to thank all my colleagues and outstanding researchers of the Crypto Team: Adrian, Alain, Anca, Angelo, Aurélien, Aurore, Céline, Dahmun, Damien, Fabrice, Florian, Geoffroy, Georg, Guiseppe, Hoeteck, Houda, Itai, Jérémy, Léo, Liz, Louiza, Michele, Michele, Pierre-Alain, Pooya, Pierrick, Rafael, Raphael, Razvan, Romain, Sonia, Sylvain, Tancrède, Thierry and Thomas.

My adventure in France would not have been the same without my friends Antonia, Cyprien, Dahmun, Frédéric, Hoeteck, Guiseppe, Liz, Pierrick, Tancrède, Thibault, Thomas and Víctor. Thank you all for those funny nights.

Also a special thanks to the ones who pushed me to follow a PhD while I was in Chile: Tomás Barros, José Miguel Piquer, Alejandro Hevia and Rossana Escobar. Your advice, help, inspiration and support will always be remembered.

I am more than grateful to my parents and brother, for supporting me all my life, for encouraging me to always be a better person, for their patience and love. To Francisco, for his support, for being there in the distance to laugh and listen to me. To my family for being there. I love you all. And of course to Antonia for being an excellent partner these last two years along with our cat-Mandú and our wookiee-dog Chewie. Every one of them have taught me more than I could have ever imagined before, and I deeply appreciate these lessons. Let us hope that we will keep enriching each other's lives for a while!

Contents

List of Algorithms

Chapter 1

Introduction

Cryptography as a science has evolved from the art of constructing and breaking codes based on creativity and skills. Modern cryptography is a science that studies techniques for securing digital information, transactions and distributed computation.

Private message exchange has been the mainstream of cryptography since its origins. This problem has been extensively studied from different points of view and scenarios leading to many protocols depending on the characteristics of the problem. Solutions for secure communications over insecure channels might differ depending on the security guarantees expected, the power of the adversary, the number of parties involved, the type of information, etc. Cryptography can be seen as an endless cat-and-mouse game, where attackers are in the constant pursuit of gaining information of schemes (hopefully) carefully designed. Designers define exactly the scope and goals for specific applications which now are much more than secret communication, but message authentication, digital signatures, protocols for exchanging secret keys, authentication protocols, electronic auctions and elections, and digital cash.

In the past, computing involved a single process executing an algorithm sequentially, but in modern computing this situation has changed. As in 2016, it is estimated that every day more than 200 million credit card payments are made and more than 5 million new devices are connected to the Internet. Without a doubt, the explosion of connected devices - enabled in part because of the reduction of prices in the hardware - has captured people's attention and it has developed the idea of Distributed Environments, which may range from a single computer to a data center distributed globally. This is the intuition behind Cloud Computing: Using external resources to help the computation of a task in a distributed and possibly cooperative way.

Encryption The primary aim of the cryptography is to communicate with parties securely over an insecure media. The basic setting is two parties sending messages over a channel which might be controlled by an adversary. To achieve this, an encryption scheme is used. It consists of a pair of algorithms. One algorithm is called encryption which is used by the sender and takes as input a message called *plaintext* and a key called *encryption key* outputting an enciphered version of the message called *ciphertext* which is sent over the channel. Upon the reception of a ciphertext, the receiver uses the second algorithm called decryption which takes as input the ciphertext and a key called *decryption key* and outputs the original message or plaintext.

In this setting, in order to achieve privacy at least the receiver must know something

that the adversary does not know. Intuitively, the security of an encryption scheme relies solely on the *possibility* (classic approach) or *feasibility* (modern approach) to extract this secret information.

Defining security formally is a delicate issue with multiple flanges. The first formal definition was given by Shannon [Sha49] known as *perfect privacy*. In this approach, the privacy is described in terms of information entropy. It states that it is impossible for an adversary with unlimited computation resources to gain any information about the plaintext other than the length of the ciphertext. In the modern definition, the ciphertext might leak information about the plaintext as long as it cannot be *efficiently* computed.

Public Key Cryptography One of the advantages of defining security in the *computational-complexity* approach is that it allows the existence of public-key encryption. The notion of public-key cryptography was introduced by Diffie and Hellman [DH76] and it is one of the major breakthroughs of modern cryptography because it allows the parties to have a secure communication without requiring previously established secrets. The public-key cryptography setting is as follows: Each user has a pair of keys (a secret one and a public one). While the public key is known by all parties, the secret key is only known by the user. The idea behind this setting is that a user that wants to encrypt a message uses the public key of the receiver to create the ciphertext. The receiver uses its secret key to decrypt the ciphertext and recover the original plaintext. The first public-key encryption system was proposed in 1978 by Rivest, Shamir and Adleman [RSA78].

Pseudorandom Number Generators Pseudo-random number generators (PRNGs) are widely used as a randomness source in cryptographic applications. A PRNG is an efficient deterministic mechanism for generating random numbers on a computer by expanding a short randomly selected seed into a much longer pseudorandom bit sequence that is computationally indistinguishable from a truly random sequence. Random numbers are in session keys, initialization vectors, public-key generation, and many other places.

Cloud Computing Cloud computing is a general term for hosted services over the Internet. Providers enable users, institutions and companies to consume computational resources on demand rather than deploying and maintaining a computer infrastructure themselves. Cloud Computing has given users multiple benefits like using computing resources on-demand, instantaneous scalability and granular level of utilization and payment.

A private cloud is a cloud stored and maintained in the private data centers of each company. This solution requires to have people to administrate the servers, additional energy costs, security measurements, etc. On the other hand, a public cloud is sold on-demand, typically by the minute or the hour. Customers only pay for the CPU cycles, storage or bandwidth they consume. Leading public cloud providers include Amazon Web Services (AWS), Microsoft Azure, IBM/SoftLayer and Google Compute Engine.

Cloud computing provides a number of advantages but it also introduces new threats. To maximize the efficiency, providers run multiple virtual machines on the same physical server, allowing disjoint customers to use the same physical hardware using the same base operating system. In particular, multiple customers can share memory, disks and processors. If an adversary escapes the isolation, the hypervisor of the virtual machine might violate the confidentiality of another customer.

1.1 This Work

Some issues are presented of a cloud computing setting where the hardware is usually outsourced and the provider gives access to one (or several) machines running an operating system. In this work we address three problems in a Cloud Computing setting.

In a cloud computing environment, the security model of a PRNG might not be realistic because an attacker, at some point, might be able to acquire the internal state. Not because of a flaw in the implementation, but because maybe the attacker managed to escape the virtual machine isolation and was capable to read the seed file from the disk. In a traditional PRNG, if the attacker acquires the internal state, she can follow all the outputs and all the updates of the internal state. This means that if the PRNG is ever attacked successfully, then it can never recover to a secure state.

The other two works presented here are applications to take advantage of a Cloud. The first one introduces a Password-protected secret sharing (PPSS) scheme that allows a user to publicly share its high-entropy secret across different servers and to later recover it by interacting with some of these servers using only his password without requiring any authenticated data. In particular, this secret will remain safe as long as not too many servers get corrupted. However, servers are not always reliable and the communication can be altered. To address this issue, we propose new robust PPSS schemes which are significantly more efficient than the existing ones. We achieve this goal in two steps. First, we propose a generic technique to build a Robust Gap Threshold Secret Sharing Scheme (RGTSSS) from any threshold secret sharing scheme. In the PPSS construction, this allows us to drop the verifiable property of Oblivious Pseudorandom Functions (OPRF). Then, we use this new approach to design two new robust PPSS schemes that are quite efficient, from two OPRFs. Both are proven in the random oracle model, just because our RGTSSS construction requires random non-malleable fingerprints. This is easily guaranteed when the hash function is modeled as a random oracle.

Finally, we propose a Searchable Symmetric Encryption (SSE) scheme which is a useful cryptographic construction for privacy protection in cloud storage solutions. It allows a client to securely outsource an encrypted database to a server that can be searched and modified.

Recent breakthroughs made SSE able to support (very) large databases and complex queries (involving more than one keyword), while keeping a high level of privacy. Also, verifiable schemes, secure against malicious servers, have also recently been designed, but they suffer from the lack of complex queries support. Pointing in that direction, we propose a black-box method to build efficient, dynamic, multiple-keyword and verifiable SSE schemes, using components like hash trees, or set hashing function. We provide a rigorous security analysis by describing the amount of information it leaks. We also give the first implementation for all these functionalities put together, and we report on experimentations of this implementation by illustrating the cost of multiple-keyword search and verification of symmetric searchable encryption. These experiments demonstrate that verifiable SSE is practical, even for large databases.

Table 1.1: Summary of our Results

PRNG	Instance	State Size	$\lambda^{(*)}$	Security Property Broken
OpenSSL	Non FIPS PRNG	8576	320	Backward Security
Android	`SHA1PRNG`	3136	0	Resilience
OpenJDK	`SHA1PRNG`	352	32	Backward Security
	`NativePRNG`	5472	1056	Robustness
Bouncycastle	`SHA1PRNG`	448	0	Resilience
IBM	`SHA1PRNG`	680	32	Backward Security

(*) λ denotes the size of the part of the internal state an attacker needs to corrupt to attack the PRNG.

1.2 Contributions

Characterization of Real-Life PRNGs under Partial State Corruption

From a theoretical viewpoint, we formally extend the security model of [DPR+13], to capture the behavior of a PRNG against an attacker that has partial access to its internal state. From a practical side, we characterize and give a new security analysis of PRNG implementations from widely used providers in real-life applications: OpenSSL, OpenJDK, Android, Bouncycastle and IBM. To our knowledge, while intensively used in practice, these PRNGs have not been evaluated w.r.t. recent security models. Our analysis reveals new vulnerabilities of these PRNG due to the implementation of their internal state in several fields that are not updated securely during PRNG operations. Our results are summarized in Table 1.1. In this table, we give the size in bits of the internal state of the PRNG and the number of bits (named λ) that an attacker needs to compromise in order to mount an attack against the PRNG.

Robust Password-Protected Secret Sharing

Our PPSS protocol follows the methodology from [JKK14]: it is based on the use of pseudorandom functions (PRFs) evaluated on the password to mask the shares of the secret. These evaluations are performed with servers that own the PRF keys, in an oblivious way, hence the so-called *oblivious pseudorandom functions* (OPRFs).

Our main contribution is the efficient realization of the robustness, that consists in a single check at the very end of the protocol, during the secret reconstruction. We point out that, in order to achieve robustness in a PPSS protocol, one does not need to distinguish between correct and incorrect shares at each individual evaluation with a server like in the verifying setting presented in [JKK14].

Actually, we propose a new efficient method to convert any Secret Sharing Scheme in a (t_ℓ, t_r, n)-Robust Gap Threshold Secret Sharing Scheme (RGTSSS) that guarantees to efficiently identify the correct values (and reconstruct the secret) if at least t_r shares are correct. However, if at most t_ℓ shares are correct, the protocol leaks no information about which shares are correct.

More precisely, we are using an encoding to generate fingerprints of the shares. The assumption that invalid shares encode into random fingerprints is enough for the recon-

struction technique to be able to select the correct shares, when their number is high enough. This can be achieved using a hash function modeled as a random oracle [BR93].

Such a (t_ℓ, t_r, n)-RGTSSS allows the user to execute the PPSS protocol in a robust way. If the number of correct servers' answers is above the threshold t_r, the user can efficiently identify the valid ones and reconstruct the secret. If the number of answers is below another threshold t_ℓ, no information about the secret is leaked. It is indeed important that not too few correct shares can be detected as correct as this could result in offline dictionary attacks. For instance, in the case where shares could be individually checked, a dishonest server could easily mount an offline dictionary attack. With our new primitive, even t_ℓ corrupted servers cannot perform an offline dictionary attack as they would still need to interact with at least one additional server. The main difference to [JKK14] is in the way to achieve robustness: We ask a bit more from the secret sharing scheme, but much less from the OPRF, allowing more efficient constructions for the latter, which highly improves on the global efficiency.

While similar to [JKKX16] in terms of server interaction efficiency, our technique takes advantage of the RGTSSS to optimize the secret reconstruction. The scheme proposed by [JKKX16] has one significant drawback: the client is supposed to specify the exact set of servers involved in the secret recovery from the beginning, which may lead to frequent failures as the servers may misbehave. Moreover, in case of such a failure, the user is unable to detect the cheating servers. To overcome this drawback when a large number of servers are involved in the protocol, our approach makes use of the *robustness* feature, to ensure the recovery of the secret and the detection of dishonest servers.

We propose two efficient OPRF constructions: The first one is based on the One-More Gap Diffie-Hellman assumption and its efficiency is quite similar to the one in [JKKX16]. Secondly, we introduce a new oblivious evaluation of the Naor-Reingold PRF [NR97], based on the sole DDH assumption. This new construction compares very favorably to other oblivious evaluations of the Naor-Reingold PRF: our protocol simply uses ElGamal encryption [ElG85] in prime order groups with simple zero-knowledge proofs, whereas for example the scheme in [JKK14] has to work in composite order groups with Paillier encryption [Pai99] and more complex zero-knowledge proofs.

By combining these building bricks, we eventually reach efficient PPSS schemes that satisfy Soundness and Robustness properties. The two proposed solutions are eventually proven in the *Random Oracle Model* (ROM) [BR93], as our RGTSSS construction requires random non-malleable fingerprints. This can be achieved by using a hash function that is modeled as a random oracle.

Verifiable Symmetric Searchable Encryption with Boolean Queries and Controlled Leakage

We present the CXT (for Composite Cross Tag) verifiable, dynamic, and multiple-keyword SSE scheme. Our SSE solution is inspired from Cash *et al.* [CJJ+13] OXT scheme, using a Single Keyword Search (SKS) scheme for the least frequent keyword with a cross matching protocol for the other keywords. However, we add verification mechanisms for both components. We thus build CXT from a verifiable SKS and a verifiable data structure allowing to test and verify membership queries.

We give a formal proof of this generic construction and we also give two instantiations of CXT : the first being very simple and efficient, and the second one being based on

ORAM-related components, and only leaking a very small amount of information. In particular, this second instantiation is forward secure.

We present an open source implementation of the simple instantiation and show that our approach is practical.

Chapter 2

Preliminaries

In this section we introduce briefly the basic ideas of probability and cryptography theory. For a complete definition we refer to [HPS08].

2.1 Random Variable.

A random variable, usually written X, is a function $X : \Omega \to \mathcal{R}$ whose domain is the sample space Ω and the possible values are numerical outcomes in $\mathcal{R}$ of a random phenomenon. Random variables are useful for defining events. For example if X is a random variable then, for example, we can define the following event: $\{\omega \in \Omega : X(\omega) \leq x\}$.

2.2 Probability Distribution.

Let $X : \Omega \to \mathcal{R}$ be a random variable. The probability distribution function of X denoted by P_X is defined to be: $P_X(x) = \Pr(X = x)$ and it can be seen as P_X being the probability that X takes on the value x. By $x \xleftarrow{\$} X$ we mean that x is sampled according to the distribution of the random variable X.

2.3 Indistinguishability.

The notion of computational indistinguishability is central to the theory of cryptography. Informally speaking, two probability distributions are computationally indistinguishable if no efficient algorithm can tell them apart (or distinguish them). Formally, two distributions X and Y are said to be (t, ε)-*computationally indistinguishable*, (which we denote $\mathbf{CD}_t(X, Y) \leq \varepsilon$), if for any distinguisher $\mathcal{A}$ running within time t, $\Pr[\mathcal{A}(X) = 1] - \Pr[\mathcal{A}(Y) = 1] \leq \varepsilon$. When $t = \infty$, meaning $\mathcal{A}$ is unbounded, we say that X and Y are ε-*close*, and their *statistical distance* is at most ε: $\mathbf{SD}(X, Y) \leq \varepsilon$.

2.4 Symmetric Cryptography.

Symmetric cryptography (also called private-key cryptography), is a setting where two parties share some secret information called a key, and use this key when they wish to communicate secretly with each other. An implicit assumption in any system using

private-key encryption is that the communicating parties have some way of initially sharing a key in a secret manner. Two users share a secret key K to communicate confidentially. Each user *encrypts* the message he wants to send to the other user, using the key K; and the other user *decrypts* the received encrypted message (a.k.a., *ciphertext*) to get the original message back, using the same key K. Anybody who intercepts the ciphertext should not be able to learn anything about the original message, without knowledge of the secret key K. Formally defined:

Definition 2.1. *A private-key encryption scheme is comprised of three algorithms:*

- *The key-generation algorithm* `Gen` *is a probabilistic algorithm that outputs a key k chosen according to some distribution that is determined by the scheme.*
- *The encryption algorithm algorithm* `Enc` *takes as input a key k and a plaintext m and outputs a ciphertext c. We denote the encryption of the plaintext m using the key k by* $\texttt{Enc}_k(m)$.
- *The decryption algorithm* `Dec` *takes as input a key k and a ciphertext c and outputs a plaintext m. We denote the decryption of the ciphertext c using the key k by* $\texttt{Dec}_k(c)$.

The basic correctness requirement of any encryption scheme is:

- *For every key k output by* `Gen` *and every plaintext m in the appropriate underlying plaintext space, it holds that:* $\texttt{Dec}_k(\texttt{Enc}_k(m)) = m$.

2.5 Asymmetric Cryptography.

In contrast to symmetric cryptography, in asymmetric cryptography (also called public-key cryptography) the two parties do not share the same secret key K but they use two different keys, a *public key* (PK) and a *secret key* (SK). An important property is that given the public key it is infeasible to forge the secret key. In this scheme, the receiver sends his public key to the sender over an insecure channel, who then encrypts the plaintext using that key and transmits the resulting ciphertext. Finally, the receiver can decrypt the ciphertext using his secret key. In Public Key Cryptosystems both users exchange their public keys publicly to encrypt a plaintext that can be decrypted only with the secret key associated with each public key. By publicly revealing PK one does not reveal an easy way to compute SK.

The definition of public-key encryptions is as follows:

Definition 2.2. *A public-key encryption scheme is defined by a tuple of probabilistic, polynomial-time algorithms* (`Gen`, `Enc`, `Dec`) *that satisfies the following:*

- *Algorithm* `Gen` *takes as input a security parameter 1^n and outputs a pair of keys* (pk, sk). *We refer to the first of these as the* public key *and the second as the* secret key.
- *Algorithm* `Enc` *takes as input a public key pk and a message m from some underlying plaintext space. It outputs a ciphertext c, and we write this as $c \leftarrow \texttt{Enc}_{pk}(m)$.*
- *Algorithm* `Dec` *takes as input a private key sk and a ciphertext c and outputs a message m or a special symbol $\perp$, and we write this as $m \leftarrow \texttt{Dec}_{sk}(m)$.*

It should satisfy the following property:

- *For every n, every* (pk, sk) *pair output by* $\texttt{Gen}(1^n)$*, and every message m in the appropriate underlying plaintext space, it holds that:* $\texttt{Dec}_{sk}(\texttt{Enc}_{pk}(m)) = m$.

2.6 Game Playing Framework.

Most of our security proofs are proofs by games (also called hybrid arguments) as defined by Shoup in [Sho01, KR01, BR06]. The idea is to define an *attack game* with procedures to respond to adversary oracle queries, then to prove security using the sequence-of-games as follows: The first game $\mathbf{G}_0$ is the original attack game with respect to a given adversary and cryptographic primitive corresponding to some security notion. The last game $\mathbf{G}_n$ corresponds to some security notion or is such that the adversary just cannot win. The general framework is that the probability of some event S_i defined in the game $\mathbf{G}_i$ is *very close* or *negligibly close to* the probability of the event S_{i+1}. In other words, we prove that two consecutive games are indistinguishable either perfectly, statistically, or computationally. In constructing such proofs, it is desirable that the changes between successive games are very small, so that analyzing the change is as simple as possible.

Chapter 3

Inside the Cloud: Random Number Generation

3.1 Introduction

Randomness is one of the fundamental requirements in cryptography. It is required in most of the fundamental tasks such as encryption, key generation, nonces, random paddings, salts, generation of initialization vectors among several others. The security of these cryptographic algorithms and protocols relies on a source of unbiased and uniformly distributed random bits. Unfortunately, generating random numbers is not that easy as it seems. Von Neumann [VNT61] said *Any one who considers arithmetical methods of producing random digits is, of course, in a state of sin. For, as has been pointed out several times, there is no such thing as a random number* referring to the fact that it is fundamentally impossible to produce truly random numbers on any deterministic device. In the purest sense of the word, the best we can hope for are pseudo-random numbers which in practice they are generally sufficient even for demanding security-critical applications.

Nevertheless, in practice it is possible to produce unpredictable bits using an algorithm called Pseudo Random Number Generation (PRNG) which accumulates entropy from the environment and outputs pseudorandom strings indistinguishable from the uniform distribution to a computationally-bounded adversary. A typical assumption at the moment of design a PRNG is that the PRNG has an internal memory in which it is possible to store information and access it to produce the output. This memory is called internal state S and it remains secret to the adversary. However, in practice, keeping a piece of memory secret might not be that easy as it is believed. The internal state can be partially compromised through memory corruption attacks such as buffer overflows or fault attacks.

Attacking the internal state is a particularly sensitive topic in a cloud setting, where the hardware is usually outsourced and the PRNG could be running on a virtualized environment. This means that the cloud provider or another user of the server could have access to the whole memory [RTSS09, IIES14], contradicting one of the PRNG design fundamental: The adversary cannot have access to the memory. Different examples of memory attacks are presented in [Erl07] and in [vdVdSCB12].

An implementation error in the software might also help an attacker to learn something secret of the memory like was the case of the CVE-2014-0160 known as Heartbleed bug [Hea14] that affected one of the most used cryptographic libraries: OpenSSL. An attacker could get access to content of the server or client memory by crafting a TLS or DTLS

heartbeat packet. Although the attacker can control the size of the compromised memory, it can not control its location, therefore it might have a total or partial access to sensitive information as the internal state of the PRNG.

Currently there are many PRNG implementations from different vendors, and most of them rely on internal directives and parameters that are poorly documented or even undocumented. In most implementations, a PRNG contains a dedicated internal state S which is *refreshed* periodically with the entropy I collected from its environment (such as network input/output, inter-interrupt timings from some interrupts, inter-keyboard timings, processor clock cycles) and secondly used to compute pseudorandom strings. The entropy collection is a hard problem and it takes much more time than the generating an output. In order to have a constant output of random strings, PRNGs typically maintains an internal state, which is the most critical part of the PRNG and therefore needs to be kept secure during its update.

3.1.1 Related Work

Random number generation does not steal the spotlight of design cryptographic primitives and protocols and yet it might be equally important in order to keep the security of algorithms and protocols. After the guidelines for developing PRNGs of Kelsey *et al.* [KSWH98] and Gutmann [Gut98] not that many implementations have been analyzed.

Theoretical cryptographers also have done some reserach in the area of PRNGs. The first work in the field was presented by Desai *et al.* [DHY02] in which they modeled a PRNG as a pair of algorithms: the *Seed Generation* algorithm and the *Output Generation* algorithm. This model assumes the existence of an entropy pool, different from the internal state, in which randomness is accumulated and it is used to refresh the internal state of the PRNG. An elegant and remarkable work of Barak and Halevi presented in [BH05] modeled a PRNG as a pair of algorithms (refresh and next) based on the design guidelines of Kelsey. Barak and Halevi defined a new security property called *robustness* that assesses the behavior of a PRNG after its internal state was compromised, but it fails to capture gradual entropy accumulation present in most real-life implementations.

A follow-up paper by Dodis *et al.* [DPR+13] identified the problem of this slow (and potentially malicious) entropy accumulation and refined the robustness property of a PRNG defined by Barak and Halevi. This new property, still named robustness, captures the idea of how the entropy of the input data should be accumulated in the internal state after a state compromise. A recent work of Dodis *et al.* [DSSW14] extends the robustness model to address the *premature next attack* where the internal state has insufficient entropy and an output is generated. To our knowledge, this last security model is the strongest one as it considers the most powerful attacker against a PRNG.

Our work complements the security model of [DPR+13] but in a different way than [DSSW14] does. Inspired on the work of Akavia, Goldwasser and Vaikuntanathan [AGV09], we consider the situation where an attacker can access partially to the memory of the PRNG. Hence we propose a new attacker profile that captures real-life situations where a partial internal state corruption is possible. We also show an analysis of real-life PRNGs using this security model and we demonstrate how it can help to identify new vulnerabilities. In particular, we show that a *full* internal state corruption is not necessary to compromise a PRNG, instead only a *partial* one may be sufficient. We characterize how a PRNG can be attacked in order to produce a predictable and we identify how many bits of the internal state are required to mount an attack against the PRNG.

Other Randomness Weaknesses. Several recent attacks occurred due to an insufficient understanding of PRNG implementations. Michaelis *et al.* in [MMS13] described and analyzed several PRNG libraries written in Java and they their weaknesses. One striking example is the failure in the Debian Linux distribution, where a commented line of code in the OpenSSL PRNG forced the only source of entropy to be the Process Identifier (PID). An analysis and an attack that breaks the forward-security of Linux PRNGs `dev/random` and `dev/urandom` was done in 2006 by Gutterman *et al.* in [GPR06]. In 2013 Dodis *et al.* [DPR+13] presents an attacks against these PRNG, but related to their internal entropy estimator. Heninger *et al.* in [HDWH12] presented an analysis of Linux's PRNG that at boot time the some SSH and TLS keys are generated with insufficient entropy. The Windows PRNG `CryptGenRandom` was analyzed in 2006 by Dorrendorf *et al.* in [DGP09] where the authors showed an attack on the forward security of the PRNG implemented in Windows 2000, for which a fix has been published. Argyros and Kiayias in [AK12] presented some practical techniques and algorithms for exploiting randomness vulnerabilities in PHP. More recently, a flaw in the Android PRNG, identified by Kim *et al.* [KHL13], has been actively exploited against Android-based Bitcoin wallets.

3.1.2 Roadmap

First, we give a formal definition of a PRNG in Section 3.2. Using recent theoretical results in the field, we provide in this work a new security model and we present it as a framework in Section 3.3. We use this framework in Section 3.4 for the analysis of widely used PRNGs and we identify new potential vulnerabilities due to the way they handle their internal state during its update step. Finally in Section 3.4.6 we give a secure implementation using our new framework.

3.2 Preliminaries

In this section we describe our notation and definitions used, adapted from the work of Dodis *et al.* [DPR+13].

Notation. We denote with $[S||I]$ the concatenation of the bit string S with the bit string I and we denote with $|S|$ the length (in bits) of the bit string S. We denote with $S[n]$ the n^{th} byte of S and $S[n, \cdots, m]$ (or $[S]_n^m$) the extracted bytes of S from the n^{th} to the m^{th}. Instructions and code references are denoted with the verbatim style as in `SecureRandom`. We denote with $\mathbf{H}_\infty(X)$ the min-entropy of a distribution X.

Extractors. Let $\mathcal{H} = \{h_X : \{0,1\}^n \to \{0,1\}^m\}_{X \in \{0,1\}^d}$ be a hash function family. We say that $\mathcal{H}$ is a (k, ε)-*extractor* if for any random variable I over $\{0,1\}^n$ with $\mathbf{H}_\infty(I) \geq k$, the distributions $(X, h_X(I))$ and (X, U) are ε-close where X is uniformly random over $\{0,1\}^d$ and U is uniformly random over $\{0,1\}^m$. We say that $\mathcal{H}$ is ρ-universal if for any inputs $I \neq I' \in \{0,1\}^n$ we have $\Pr_{X \xleftarrow{\$} \{0,1\}^d}[h_X(I) = h_X(I')] \leq \rho$. To construct a randomness extractor we use the Leftover-Hash Lemma [ILL89].

Lemma 3.1 (Leftover-Hash Lemma)**.** *Assume that $\mathcal{H}$ is ρ-universal where $\rho = (1+\alpha)2^{-m}$ for some $\alpha \geq 0$. Then, for any $\gamma > 0$, it is also a (γ, ε)-extractor for $\varepsilon = \frac{1}{2}\sqrt{2^{m-\gamma} + \alpha}$.*

Proof. Fix any $I \neq I' \in \{0,1\}^n$, with $\mathbf{H}_\infty(I)$ and $\mathbf{H}_\infty(I') \geq k$, $X, X' \in \{0,1\}^d$ and first consider the statistical distance between $(X, h_X(I))$ and (X, U):

$$\begin{aligned}\mathbf{SD}((X, h_X(I)), (X, U)) &= \frac{1}{2}\|(X, h_X(I)) - (X, U)\|_1 \\ &\leq \frac{1}{2}\sqrt{2^d \cdot 2^m} \cdot \|(X, h_X(I)) - (X, U)\|_2\end{aligned}$$

Consider $\|(X, h_X(I)) - (X, U)\|_2$. By definition, $\Pr_{X,I}[(X, h_X(I)) = (X', h_{X'}(I'))] = \|(X, h_X(I))\|_2$ and $\|(X, h_X(I))\|_2 = \|(X, h_X(I)) - (X, U)\|_2 + 2^{-m-d}$, hence:

$$\|(X, h_X(I)) - (X, U)\|_2 = \Pr_{X,I}[(X, h_X(I)) = (X', h_{X'}(I'))] - 2^{-m-d}.$$

As $\mathbf{H}_\infty(I) \geq k$ and $\mathbf{H}_\infty(I') \geq k$ and $\mathcal{H}$ is $2^{-m} \cdot (1+\alpha)$-universal:

$$\begin{aligned}\Pr_{X,I}[(X, h_X(I)) = (X', h_{X'}(I'))] &\leq \Pr_X[X = X'] \cdot (\Pr_I[I = I'] + \Pr_X[I \neq I' \mid h_X(I) = h_X(I')]) \\ &\leq 2^{-d} \cdot (2^{-k} + 2^{-m} \cdot (1+\alpha))\end{aligned}$$

Finally, with $\alpha = 4 \cdot \varepsilon^2 - 2^{m-\gamma}$:

$$\begin{aligned}\mathbf{SD}((X, h_X(I)), (X, U)) &\leq \frac{1}{2} \cdot \sqrt{2^d \cdot 2^m} \cdot \sqrt{\frac{4 \cdot \varepsilon^2}{2^d \cdot 2^m}} \\ &\leq \varepsilon\end{aligned}$$

Following, $\mathcal{H} = \{h_X : \{0,1\}^n \to \{0,1\}^m\}_{X \in \{0,1\}^d}$, is a (γ, ε)-extractor for $\varepsilon = \frac{1}{2}\sqrt{2^{m-\gamma} + \alpha}$. □

If $\alpha = 0$, then $\varepsilon = \frac{1}{2}\sqrt{2^{m-\gamma}}$ and the constraint on γ and m is $2 + 2\log(\varepsilon) = m - k$, or $k = m + 2\log(1/\varepsilon) - 2$. In addition, if $\alpha \neq 0$, then $\alpha + 2^{m-\gamma} = 4 \cdot \varepsilon^2$ and the condition is satisfied if $\alpha = 2 \cdot \varepsilon^2$ and $\alpha = 2 \cdot \varepsilon^2$. Then for various values of α, we have the following constraints on k, m and n:

- if $\alpha = d \cdot 2^{m-n}$, $n = m + 2\log(1/\varepsilon) + \log(d) - 1$ (these are the bounds used in [DPR+13]).
- if $\alpha = d \cdot 2^{\lambda+m-n}$, $n = \lambda + m + 2\log(1/\varepsilon) + \log(d) - 1$ (these are the bounds used in this work).

Deterministic Pseudorandom Generators. We say that a function $\mathbf{G} : \{0,1\}^m \to \{0,1\}^m$ is a (deterministic) (t, ε)-*pseudorandom generator* (PRG) if $\mathbf{CD}_t(\mathbf{G}(\mathcal{U}_m), \mathcal{U}_n) \leq \varepsilon$.

Pseudorandom Number Generator. We recall the definition of a PRNG given in [DPR+13]. It uses the following notations: a state $S \in \{0,1\}^n$, an input $I \in \{0,1\}^p$, an output $R \in \{0,1\}^\ell$.

Definition 3.2 (PRNG)**.** *A PRNG is a triple of algorithms* $\mathcal{G} = (\mathsf{setup}, \mathsf{refresh}, \mathsf{next})$ *where:*

- setup*: A probabilistic algorithm that outputs some public parameter* seed*.*
- $\mathsf{refresh}$*: A deterministic algorithm that, given* seed*, a state* S *and an input* I*, outputs a new state* $S' = \mathsf{refresh}(S, I, \mathsf{seed}) \in \{0,1\}^n$*.*

- next*: A deterministic algorithm that, given* seed *and a state* S*, outputs a pair* $(S', R) =$ $\mathsf{next}(S, \mathsf{seed})$ *where* S' *is the new state and* R *is the output.*

The parameter seed is public and common to all the above algorithms; for clarity, we remove it and we write $S' = \mathsf{refresh}(S, I)$ instead of $\mathsf{refresh}(S, I; \mathsf{seed})$ and $(S', R) = \mathsf{next}(S)$ instead of $\mathsf{next}(S; \mathsf{seed})$.

Distribution Sampler As in [DPR+13], we divide the adversary into two parts, the first one is the *adversary* $\mathcal{A}$ whose goal is to distinguish the outputs of the PRNG from random and the second one is the *distribution sampler* $\mathcal{D}$ which is used by $\mathcal{A}$ to produce (potentially biased) inputs.

Definition 3.3 (Distribution Sampler)**.** *A distribution sampler* $\mathcal{D}$ *is a stateful and probabilistic algorithm which, given the current state* σ*, outputs a tuple* (σ', I, γ, z) *where:*

- σ' *is the new state for* $\mathcal{D}$*;*
- $I \in \{0,1\}^p$ *will be the next input for the* refresh *algorithm;*
- γ *is some* entropy estimation *of* I*;*
- z *is the possible* leakage *about* I *given to* $\mathcal{A}$*.*

We denote $q_{\mathcal{D}}$ the upper bound on the number of executions of $\mathcal{D}$. The distribution sampler $\mathcal{D}$ is provided by the adversary $\mathcal{A}$ and its goal is to generate the inputs that will be used by $\mathcal{G}$ to improve the quality of its entropy with the refresh algorithm. The adversary has to provide the definition of the distribution sampler before its knowledge of the seed. The distribution sampler models the potentially adversarial environment of $\mathcal{G}$, with biased inputs. A distribution sampler is called *legitimate* if the min-entropy of every input I_j is not smaller than the entropy estimate γ_j, even given all the additional information: $\mathbf{H}_\infty(I_j \mid I_1, \ldots, I_{j-1}, I_{j+1}, \ldots, I_{q_{\mathcal{D}}}, z_1, \ldots, z_{q_{\mathcal{D}}}, \gamma_1, \ldots, \gamma_{q_{\mathcal{D}}}) \geqslant \gamma_j$, for all $j \in \{1, \ldots, q_{\mathcal{D}}\}$ where $(\sigma_i, I_i, \gamma_i, z_i) = \mathcal{D}(\sigma_{i-1})$ for $\sigma_0 = 0$ and $i \in \{1, \ldots, q_{\mathcal{D}}\}$.

3.3 PRNG Security

In this section, we introduce our new security model. First we analyze existing security models and we point out details that were not taking in account. Finally, we illustrate our analysis on a concrete example and we give the formalism.

3.3.1 Theory and Practice

From Security Models to Implementations. We discuss briefly some interesting common points in the security models presented in [DHY02, BH05, DPR+13] as well as their potential use to assess PRNG implementations. Three security models consider an adversarial environment for the PRNG. The security model of [DHY02] does not take into account an attack in which the PRNG is refreshed with adversarial inputs, whereas this situation is considered in [BH05] and [DPR+13]. In [DHY02], the internal state of the PRNG is composed of two parts, named *key* and *initial state*; the generation algorithm takes as input both of them and updates the initial state. In concrete implementations the internal state is considered as a single entity, as modeled in [BH05] and [DPR+13]. Finally,

entropy accumulation in the internal state is modeled clearly only in [DPR+13]. Therefore we use the security model of [DPR+13] as a starting point for our analysis. Furthermore, our code source analysis shows that in certain situations, only a partial compromise of the internal state is necessary to make the PRNG predictable. As a partial compromise of the internal state is not captured by any of these security models, we propose a slight modification of the security model of [DPR+13] to capture this new adversarial potential. Our modification allows to identify precisely the part of the internal state that needs to be compromised to make the output of the PRNG predictable.

From Implementations To Security Models. Definition 3.2 describes a PRNG as a triple of algorithms $\mathcal{G} = (\mathsf{setup}, \mathsf{refresh}, \mathsf{next})$, where setup is a probabilistic algorithm that outputs a public parameter seed for the generator. As entropy needs to be extracted from the inputs used to refresh the PRNG, a *randomness extractor* is needed, ensuring that each input actually gives entropy to the PRNG. However, it is well known that no deterministic extractor can extract good randomness from all entropy sources and therefore a *seeded extractor* is necessary (see for example [Tre01]). The parameter seed used in the security model is the seed of the extractor, which is completely public (the only requirement is that it is random). None of the PRNG implementations use an explicit extractor: all of them use the $\mathsf{SHA1}$ function to mix new input into the current internal state or to generate outputs. We therefore assume for our analysis that the $\mathsf{SHA1}$ function defines a hash functions family used as an *extractor*, whose *seed* is the public parameter $\mathsf{K} = \mathsf{K}_0||\mathsf{K}_1||\mathsf{K}_2||\mathsf{K}_3$, where $\mathsf{K}_0 = \mathsf{5A827999}$, $\mathsf{K}_1 = \mathsf{6ED9EBA1}$, $\mathsf{K}_2 = \mathsf{8F1BBCDC}$, and $\mathsf{K}_3 = \mathsf{CA62C1D6}$ are the round constants defined in the specification [SHA95]. Hence, for all PRNGs presented in this work, we assume that the algorithm setup always outputs this public parameter K, of size 128 bits and the underlying extractor is the hash function family defined in the specification [SHA95], indexed by the parameter K. We will therefore refer to the $\mathsf{SHA1}$ function in our description as H_K, to identity the underlying hash function family. As a consequence, this assumption shows that our attacks on PRNGs are independent of the hash function used and are related to their design.

All implementations contain instructions that can be easily related to the $\mathsf{refresh}$ and next algorithms. However, while our security model considers PRNGs that may be refreshed with potentially biased inputs, in most applications, the $\mathsf{refresh}$ algorithm is called just one time with a single input. Hence after this single call, the entropy contained in S (named hereafter γ^*) is bounded by the size of the input (named hereafter p). An attacker may gain information about the behavior of the environment and estimate the entropy of this single input (named hereafter c) when collected by the PRNG. An example of this idea is presented in [MMS13], where it was discovered that the input in the Android `SHA1PRNG` implementation actually contains very low entropy since it was not generated by several calls to system variables. During our analysis, we discovered vulnerabilities that are complementary to this work, as we focus on the global behavior of the PRNG.

3.3.2 An Illustrative Example

Let us illustrate our analysis. In our security model, an attacker can compromise the internal state (partially or totally) and the PRNG security game ensures that enough entropy is accumulated in the internal state to generate output. The OpenSSL PRNG has an internal state of size 1072 bytes, which contains an entropy pool of size 1023 bytes and internal counters. The structure of S is named hereafter its *decomposition*, which is

public for OpenSSL and known to the attacker. We show that an attacker only needs to compromise 40 bytes of the internal state and to control 23 bytes of an input of size 1023 bytes (with a legitimate distribution sampler, as described hereafter in Definition 3.3) to predict a future output of the PRNG. Hence, this shows that OpenSSL PRNG does not resist a single internal state compromise.

3.3.3 The Security Model

As explained in Section 3.3.1, we propose a slight modification of the *robustness* security model of [DPR+13] to identify exactly the part of S that an attacker needs to compromise to attack a PRNG. To formalize this idea, we consider the internal state as a concatenation of several binary strings (named hereafter its *decomposition*). We model the adversarial capacity of an attacker $\mathcal{A}$ with two new functions named $\mathcal{M}$-get and $\mathcal{M}$-set that allow $\mathcal{A}$ to set or get a part of the internal state of the PRNG defined with a *mask* $\mathcal{M}$. We assume that the attacker $\mathcal{A}$ knows the *decomposition* of S and is able to choose $\mathcal{M}$. The only differences between our security game and [DPR+13] is that we replace the procedures get-state and set-state, with new procedures $\mathcal{M}$-get-state and $\mathcal{M}$-set-state, allowing to the attacker to get/set a part the internal state identified by the mask.

proc. initialize($\mathcal{D}$)
seed $\xleftarrow{\$}$ setup;
$\sigma \leftarrow 0$;
$S \xleftarrow{\$} \{0,1\}^n$;
$c \leftarrow n$;
compromised $\leftarrow$ true;
$b \xleftarrow{\$} \{0,1\}$;
OUTPUT seed

proc. finalize(b^*)
IF $b = b^*$ RETURN 1
ELSE RETURN 0

proc. $\mathcal{D}$-refresh
$(\sigma, I, \gamma, z) \xleftarrow{\$} \mathcal{D}(\sigma)$
$S \leftarrow$ refresh(S, I)
$c \leftarrow c + \gamma$
IF $c \geq \gamma^*$,
  compromised $\leftarrow$ false
OUTPUT (γ, z)

proc. next-ror
$(S, R_0) \leftarrow$ next(S)
IF compromised $=$ true,
  $c \leftarrow 0$
  RETURN R_0
ELSE
  $R_1 \xleftarrow{\$} \{0,1\}^\ell$
  RETURN R_b

proc. $\mathcal{M}$-set-state(S, M, J)
$S \leftarrow \mathcal{M}$-set(S, M, J)
$c \leftarrow c - \lambda$
IF $c < \gamma^*$,
  compromised $\leftarrow$ true
  $c \leftarrow 0$

proc. $\mathcal{M}$-get-state(S, J)
$c \leftarrow c - \lambda$
IF $c < \gamma^*$,
  compromised $\leftarrow$ true
  $c \leftarrow 0$
OUTPUT $\mathcal{M}$-get(S, J)

Figure 3.1: Procedures in Security Game MROB(γ^*, λ)

Definition 3.4 (Decomposition)**.** *A decomposition of a binary string $S \in \{0,1\}^n$ is a sequence of disjoint binary strings $(S_1, \cdots, S_k)$, such that $S = [S_1||\cdots||S_k]$. Two binary strings S and M have the* same *decomposition if $|S| = |M|$, $M = [M_1||\cdots||M_k]$ and $|S_i| = |M_i|$ for $i \in \{1, \cdots, k\}$.*

Definition 3.5 ($\mathcal{M}$-get / $\mathcal{M}$-set)**.** *The function $\mathcal{M}$-set takes as input a triple (S, M, J), where $S, M \in \{0,1\}^n$ have the same decomposition $S = [S_1||\cdots||S_k], M = [M_1||\cdots||M_k]$ and $J \subset \{1, \cdots, k\}$, then $\mathcal{M}$-set$(S, M, J) = S$, where $S_j = M_j$, for $j \in J$. The function $\mathcal{M}$-get takes as input a couple (S, J), where $S = [S_1||\cdots||S_k]$ and $J \subset \{1, \cdots, k\}$, then $\mathcal{M}$-get$(S, J) = \{S_j\}$, for $j \in J$.*

We denote q_S the upper bound on the number of executions of these functions. These functions are adversarially provided, and their goal is to let $\mathcal{A}$ choose the mask $\mathcal{M}$ over the internal state. Note that if the mask is too large (so that $\mathcal{G}$ becomes insecure), the security game will require that new input is collected. They model partial memory corruption of the PRNG.

Security Model. We now describe our security model. It is adapted from the security game $\mathsf{ROB}(\gamma^*)$ of [DPR+13] that defines the *robustness* of a PRNG. We describe briefly the parameters of the security game:

- Integer γ^*: Defines the minimum entropy that is required in S for the PRNG to be secure.
- Integer c: Defines the estimate of $\mathcal{A}$ of the amount of collected entropy.
- Integer $\lambda \leq n$: Defines the size of the mask $\mathcal{M}$.
- Boolean flag compromised: It is set to true if $c < \gamma^*$ and false otherwise.
- Boolean b: Used to challenge the attacker $\mathcal{A}$.

Our security game uses procedures described in Figure 3.1. The procedure initialize sets the parameter seed with a call to algorithm setup, the internal state S of the PRNG, as well as parameters c and b. Note that we initially set c to n and S to a random value, to avoid giving any information on S to the attacker $\mathcal{A}$. After all oracle queries, $\mathcal{A}$ outputs a bit b^*, given as input to the procedure finalize, which compares the response of $\mathcal{A}$ to the challenge bit b. The other procedures are defined below:

- Procedure $\mathcal{D}$-refresh: $\mathcal{A}$ calls the distribution sampler $\mathcal{D}$ for a new input and uses this input to refresh $\mathcal{G}$. The estimated entropy given by $\mathcal{D}$ is used by the procedure to update the counter c ($c \leftarrow c + \gamma$) and if $c \geq \gamma^*$, then the flag compromised is set to false.
- Procedure $\mathcal{M}$-set-state: Used by $\mathcal{A}$ to set part of S. First $\mathcal{A}$ calls function $\mathcal{M}$-set to update part of the internal state. Then the counter is decreased with λ, the size of the mask $\mathcal{M}$ ($c \leftarrow c - \lambda$) and as in the initial set-state procedure, if $c < \gamma^*$, the flag compromised is set to true.
- Procedure $\mathcal{M}$-get-state : Used used by $\mathcal{A}$ to get part of S. First $\mathcal{A}$ calls the function $\mathcal{M}$-get. Then the counter is decreased with λ, the size of the mask $\mathcal{M}$ ($c \leftarrow c - \lambda$) and as in the initial get-state procedure, if $c < \gamma^*$, the flag compromised is set to true.
- Procedure next-ror: It challenges $\mathcal{A}$ on its capability to distinguish the output of $\mathcal{G}$ from random, where the real output (R_0) of $\mathcal{G}$ is obtained with a call to algorithm next and the random string (R_1) is generated by the challenger. Attacker $\mathcal{A}$ responds to the challenge with a bit b^*

The Security of a PRNG under partial or total state corruption is given in Definition 3.6.

Definition 3.6 (Security of a PRNG Under Partial or Total State Corruption)**.** *A PRNG* $\mathcal{G} = (\mathsf{setup}, \mathsf{refresh}, \mathsf{next})$ *is called* $(T, \lambda, \gamma^*, \varepsilon)$-robust (resp. resilient, forward-secure, backward-secure) under partial state corruption*, with* $T = (t, q_\mathcal{D}, q_R, q_S)$*, if for any attacker* $\mathcal{A}$ *running in time at most* t*, making at most* $q_\mathcal{D}$ *calls to* $\mathcal{D}$-refresh*,* q_R *calls*

to next-ror *and* q_S *calls to* $\mathcal{M}$-get-state *or* $\mathcal{M}$-set-state*, and any* legitimate *distribution sampler* $\mathcal{D}$ *inside the* $\mathcal{D}$-refresh *procedure, the advantage of* $\mathcal{A}$ *in game* $\mathsf{MROB}(\gamma^*, \lambda)$ *(resp,* $\mathsf{MRES}(\gamma^*)$*,* $\mathsf{MFWD}(\gamma^*, \lambda)$*,* $\mathsf{MBWD}(\gamma^*, \lambda)$*) is at most* ε*, where:*

- $\mathsf{MROB}(\gamma^*, \lambda)$ *is the unrestricted game where* $\mathcal{A}$ *is allowed to make the above calls and to corrupt* λ *bits of* S*.*
- $\mathsf{MRES}(\gamma^*)$ *is the restricted game where* $\mathcal{A}$ *makes no calls to* $\mathcal{M}$-get-state/$\mathcal{M}$-set-state *(i.e.,* $q_S = 0$ *and* $\lambda = 0$*).*
- $\mathsf{MFWD}(\gamma^*, \lambda)$ *is the restricted game where* $\mathcal{A}$ *makes no calls to* $\mathcal{M}$-set-state *and a single call to* $\mathcal{M}$-get-state *(i.e.,* $q_S = 1$*) which is the very* last *oracle call* $\mathcal{A}$ *is allowed to make to corrupt* λ *bits of* S*.*
- $\mathsf{MBWD}(\gamma^*, \lambda)$ *is the restricted game where* $\mathcal{A}$ *makes no calls to* $\mathcal{M}$-get-state *and a single call to* $\mathcal{M}$-set-state *(i.e.,* $q_S = 1$*) which is the very* first *oracle call* $\mathcal{A}$ *is allowed to make to corrupt* λ *bits of* S*.*

Hence, resilience protects the security of the PRNG when it is not corrupted against arbitrary distribution samplers $\mathcal{D}$, forward security protects past PRNG outputs in case the state S gets compromised (partially or totally), backward security security ensures that the PRNG can successfully recover from state compromise (partial or total), provided enough fresh entropy is injected into the system, robustness ensures security against arbitrary combinations of the above.

3.4 Security of Real-Life PRNGs

In this section we present our analysis of PRNG implementations from different providers: OpenSSL, Android, OpenJDK, Bouncycastle and IBM. The setup algorithm for all PRNG implementations outputs the 128-bit value K, which indexes the hash function family H_K. Notice that for all games, the total number of calls is bounded polynomially by the security parameter T described in Definition 3.6.

3.4.1 Analysis of OpenSSL PRNG

The OpenSSL cryptographic library contains a PRNG which collects entropy from system calls. It has been first analyzed by Gutmann in 1998 [Gut98]; since then no new analysis has been made. It is implemented in the source file /crypto/rand/md_rand.c, as part of the OpenSSL library. The PRNG takes inputs of any size and generates outputs of size 10 bytes. The PRNG is different depending on a choice made when building the library. This choice depends on an internal parameter named `MD_DIGEST_LENGTH`, which depends on the underlying hash function used. The hash function is chosen with a dedicated flag (`USE_MD5_RAND` for the MD5 function, or `USE_SHA1_RAND` for the SHA1 function), which is by default `USE_SHA1_RAND`. Hence depending on the environment, the size of S_3 is equal to 16 bytes or 20 bytes. We assume that the SHA1 function is used in our descriptions, hence we will refer to the hash functions family H_K described before. We verified that our attack can be easily adapted if `USE_MD5_RAND` is chosen.

Internal State Decomposition. The internal state of the PRNG is implemented with five fields: `state_index`, of size 32 bits, `state`, of size 1043 bytes, `md`, of size 20 bytes, `md_count_0`, `md_count_1`, each of size 64 bits. Hence the decomposition of the internal state is given by $S = (S_1, S_2, S_3, S_4, S_5)$, where S_1, S_2, S_3, S_4, S_5 stand respectively for `state_index`, `state`, `md`, `md_count_0`, `md_count_1`. The total size of the internal state is 8576 bits and the PRNG uses this decomposition as follows: field S_1 is used as an index to select bytes in S_2; S_2 and S_3 are used to collect entropy; S_4 and S_5 are counters used during PRNG operations.

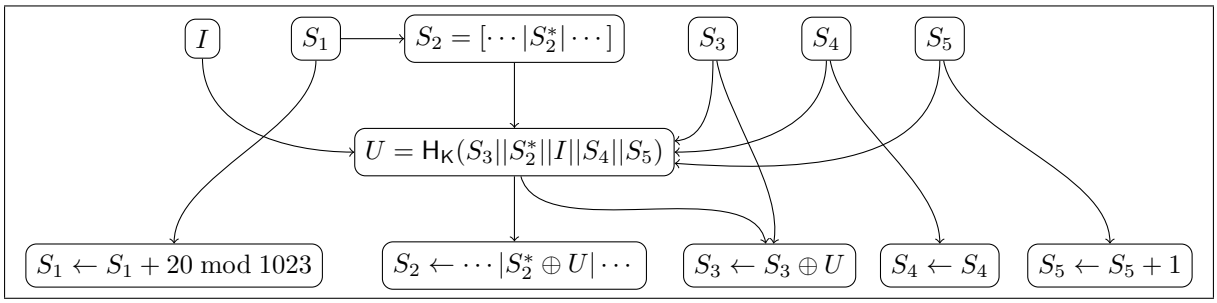


Figure 3.2: Openssl PRNG refresh Algorithm

The refresh Algorithm. This algorithm is implemented with the instruction `ssleay_rand_add`. It is fully described in Algorithm 0 and in Figure 3.2. It takes as input the current internal state $(S_1, S_2, S_3, S_4, S_5)$ and an input I of any size that is processed by blocks of 20 bytes. Starting with a 20-bytes block of S_2 that is indexed by S_1, successive blocks of S_2 are mixed with successive blocks of I. The mixing operation involves the hash functions family $\mathsf{H_K}$. This mixing operation also involves S_3, S_4 and S_5, where S_5 is incremented for each block. When this mixing is finished, the field S_3 is xor-ed with the last calculated hash. Hence after a refresh operation, $|I|$ bits of S_2 are modified, S_3 is modified, S_1 and S_5 are incremented and S_4 is not modified.

Algorithm 1 OpenSSL PRNG refresh

Require: $S = (S_1, S_2, S_3, S_4, S_5)$, I
Ensure: S'
  while $|I| > 0$ **do**
    $S_2^* = S_2[S_1 \bmod 1023, \ldots, S_1 + 20 \bmod 1023]$
    $U = \mathsf{H_K}([S_3||S_2^*||I||S_4||S_5])$
    $S_2^* = S_2^* \oplus U$
    $S_1 = S_1 + 20 \bmod 1023$
    $S_5 = S_5 + 1$
    $I = I \setminus [I]_0^{19}$
  end while
  $S_3 = S_3 \oplus U$
  return $S' = (S_1, S_2, S_3, S_4, S_5)$

The next Algorithm. This algorithm is implemented in ssleay_rand_bytes. It is fully described in Algorithm 0 and in Figure 3.3. It takes as input the current internal state

$(S_1, S_2, S_3, S_4, S_5)$, mixes S_2, S_3, S_4 and S_5 together to produce the 10-byte output R and updates S_3. Only 10 bytes from S_2 are modified, that are selected using field S_1, which behaves as an index for this operation. A second mixing operation involves S_3, S_4 and S_5 to update S_3. Hence S_2 is modified sequentially by blocks of 10 bytes with successive next calls, while S_3 is completely modified, S_1 and S_4 are incremented and S_5 is not changed. As for the refresh algorithm, the two mixing operations involve the hash function family H_K.

Note that directive `ssleay_rand_bytes` takes as input an array named `buf` which is filled with the generated output, but whose content is also used as input (referenced as I in the description below). In addition, the next algorithm uses as input the current system PID and the system time. The system PID is obtained with a call to directive `getpid`, system time is obtained froma call to directive `time`, and from a call to directive `gettimeofday` (for simplicity, we refer to these two calls as "Time" in the description of the PRNG). These inputs during the next algorithm are not explicitly compliant with the security model that requests a strict separation between the input collection and the generation, but we mention it for completeness of the description. These calls have been explicitly set by OpenSSL community to prevent a vulnerability related to a call to the `fork` function that uses a common PID for two next calls. This vulnerability is described in [Ope13].

Algorithm 2 OpenSSL PRNG next

Require: $S = (S_1, S_2, S_3, S_4, S_5)$
Ensure: S', R
$S_2^* = S_2[S_1 \bmod 1023, \cdots S_1 + 10 \bmod 1023]$
$V = \mathsf{H}_\mathsf{K}([\mathsf{PID}||\mathsf{Time}||S_3||S_4||S_5||I||S_2^*])$
$S_2^* = S_2^* \oplus V[0, \cdots, 9]$
$R = V[10, \cdots, 19]$
$S_3 = \mathsf{H}_\mathsf{K}([S_4||S_5||V||S_3])$
$S_1 = S_1 + 10 \bmod 1023$
$S_4 = S_4 + 1$
return $S' = (S_1, S_2, S_3, S_4, S_5), R$

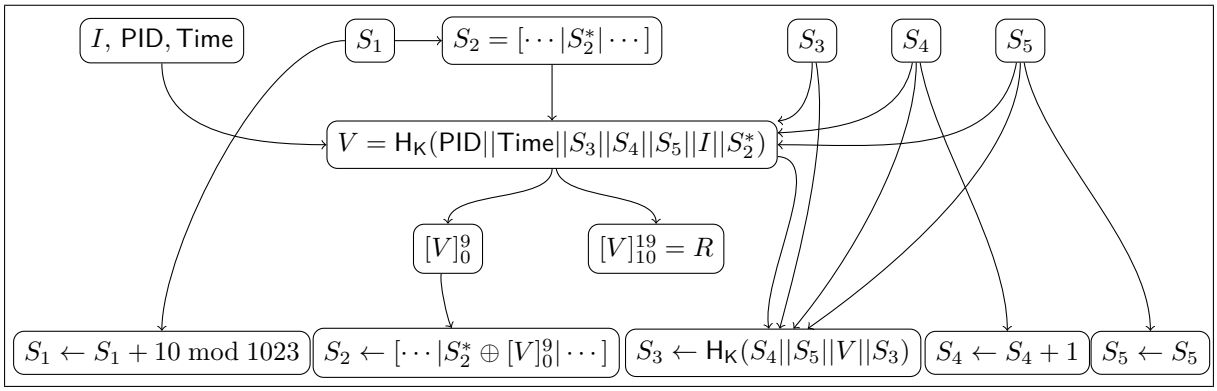


Figure 3.3: Openssl PRNG next Algorithm

Note on the Debian/OpenSSL Bug. The description of Algorithms refresh and next allow us to give a new explanation of the Debian/OpenSSL bug. In 2008, Luciano Bello

discovered that a part of the source code was commented in the OpenSSL PRNG that caused the only source of entropy to be the system PID. The commented code was concerning the calculation of $U = \mathsf{H}_\mathsf{K}([S_3||S_2^*||I||S_4||S_5])$, and especially the input I, in the string $[S_3||S_2^*||I||S_4||S_5]$ which became $[S_3||S_2^*||S_4||S_5]$. Hence the input was not taken into account in the refresh algorithm, and the only random value which was used to modify the internal state was given with the calculation of $V = \mathsf{H}_\mathsf{K}([\mathsf{PID}||S_3||S_4||S_5||S_2^*])$ in algorithm next.

Attack. We mount an attack against OpenSSL PRNG, that is based on the internal state decomposition and the fact that this state is only partially updated by the refresh and next algorithms. Our attack uses the field S_3, which is implemented with `md` and the field S_2, which is implemented with `state`. As described in Algorithms 0 and 0, when the PRNG is refreshed, the field S_3 is updated with the *last* calculated hash, whereas it is used as the entropy source for the output of the PRNG with 10 bytes of S_2. Suppose now that one uses an input of size 1023 bytes (which is the size of S_2 – or `state`) where the first 20 bytes and the last 3 bytes are 0, to refresh the PRNG. Clearly this input is independent of the parameter seed and it is therefore legitimate to use it to refresh the PRNG in our security model. Suppose now that one asks for an output. This output, which only relies on the first 10 bytes of S_2 and on S_3, is predictable. Theorem 3.7 gives the technical details of the attack. This attack is related to the refresh function that mixes new entropy sequentially by blocks of 20 bytes in the internal state, and to the next function that also reads sequentially the internal state by blocks to produce new outputs. If a block is compromised and if the attacker controls the exact block of the input that will be mixed with the compromised block of the internal state, the output is predictable. Hence the attack points a design error of the PRNG, because this behavior should not be possible.

Theorem 3.7. *Openssl PRNG is not* backward secure. *To mount an attack against the PRNG, $\mathcal{A}$ needs to corrupt 40 bytes of the internal state.*

Proof. Define the 1023-byte distribution $\mathcal{D}$. On input a state i, $\mathcal{D}$ updates its state to $i+1$ and outputs a 1023-byte input I^i: $(i+1; [I_0^i, \cdots, I_{1022}^i]) \leftarrow \mathcal{D}(i)$; where $I_0^0 = \cdots = I_{19}^0 = 0$, $I_{1019}^0 = \cdots = I_{1022}^0 = 0$ and all other bytes are random (*i.e.* $\mathcal{D}$ is legitimate with $\gamma_i = 8000$). Define the mask $M = [M_1, M_2, M_3, M_4, M_5]$, where $M_1 = 0$, $[M_2]_0^{19} = 0$, $M_3 = 0$, $M_4 = 0$, $M_5 = 0$ and $J = \{2, 3\}$ (*i.e.* this mask will be used to set the first 20 bytes of S_2 and S_3 to 0). Consider an adversary $\mathcal{A}$ against the security of the PRNG that chooses the distribution $\mathcal{D}$, and that makes the following oracle queries in the security game MBWD: one $\mathcal{M}$-set-state with S, J and M, one $\mathcal{D}$-refresh with I^0, one next-ror. Then (following refresh and next algorithm notations):

- After $\mathcal{M}$-set-state, $S_1 = 0$, $[S_2]_0^{19} = 0^{19}$, $[S_2]_{20}^{1023}$ is random, $S_3 = 0$, $S_4 = 0$, $S_5 = 0$.
- After $\mathcal{D}$-refresh, $S_1 = 0$, $[S_2]_0^{19} = \mathsf{H}_\mathsf{K}([0||0||0||0||0])$, $[S_2]_{20}^{1023}$ is random, $S_5 = 51$, $S_3 = \mathsf{H}_\mathsf{K}([0||0||0||0||51])$, $S_4 = 0$.
- After next-ror, $V = \mathsf{H}_\mathsf{K}(\mathsf{PID}||\mathsf{Time}||S_3||0||51||[S_2]_0^{19})$, $R = V_{10}^{19}$, $S_1 = 10$, $S_4 = 1$, $S_3 = \mathsf{H}_\mathsf{K}(0||51||V||\mathsf{H}_\mathsf{K}(0||0||0||0||51))$.

In this last next-ror-oracle query, $\mathcal{A}$ obtains a 10-bytes string that is predictable as it only relies on PID and Time, whereas this event should occur with probability 2^{-80}. Therefore $\mathcal{A}$ can distinguish an output of OpenSSL PRNG from random in the game $\mathsf{MBWD}(\gamma^*, 320)$, for all $\gamma^* \leq 8000$ and this PRNG is not backward secure. □

3.4.2 Analysis of Android `SHA1PRNG`

In the Android system, a full Java implementation is provided, as part of the package `security.provider. crypto`, named `SHA1PRNG`. It has been analyzed by Michaelis *et al.* in [MMS13], where the authors identified an implementation weakness that causes the internal state to be overwritten by predictable values, decreasing its entropy to 64 bits. This PRNG was also debated intensively recently, due to a weakness in its initial seeding that caused a flaw in Bitcoin wallets. This weakness caused the Android community to propose a fix to the PRNG, that simply consists in replacing it by the OpenSSL PRNG, analyzed in Section 3.4.1. Full details about the vulnerability and the proposed fix are given in [And13]. The PRNG is implemented with the class `SHA1PRNG_Secure RandomImpl` and is an inheritance from the one included in the library Apache Harmony from the package `org.apache.harmony`. It follows the method named "expansion of source bits" of IEEE standard P.1363 [Kal97].

Internal State Decomposition. The internal state of the PRNG is implemented with the fields `seed`, of size 348 bytes, and `counter`, of size 8 bytes (many other fields are used, but they are not useful to understand the PRNG operations). Hence the decomposition of the internal state is $S = (S_1, S_2)$, where S_1, S_2 stand for `seed` and `counter` and the total size of the internal state is 3136 bits. The PRNG uses this decomposition as follows: S_1 contains the collected entropy and a hash of the collected entropy; S_2 contains a counter which is incremented at each output.

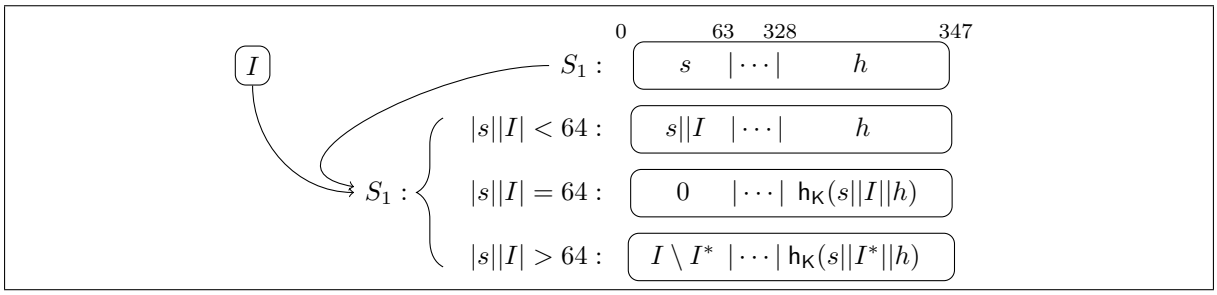


Figure 3.4: Android PRNG refresh Algorithm

Algorithm 3 Android `SHA1PRNG` refresh

Require: $S = (S_1, S_2) = ([s, \cdots, h], S_2), I$
Ensure: S'
  if $|s||I| < 64$ **then**
    $S_1[0, \cdots, 63] = [s||I]$
  end if
  if $|s||I| = 64$ **then**
    $S_1[0, \cdots, 63] = 0$, $S_1[328, \cdots, 347] = \mathsf{h}_K(s||I||h)$
  end if
  if $|s||I| > 64$ **then**
    $S_1[0, \cdots, 63] = I \setminus I^*$, $S_1[328, \cdots, 347] = \mathsf{h}_K(s||I^*||h)$
  end if
  return $S' = (S_1, S_2)$

The refresh Algorithm . This algorithm is described in Algorithm 3 and in Figure 3.4. It takes as input the current internal state (S_1, S_2), an input I of any size and updates the internal state with I. It is implemented with method `engineSetSeed` as follows: the first 64 bytes of S_1 collect the consecutive inputs and the last 20 bytes of S_1 contains a hash value. Two sub-functions are used, implemented with `SHA1Impl.updateHash` and `SHA1Impl.computeHash`. Note that these two functions correspond respectively to the *update* of the internal state of $\mathsf{H_K}$ and a function $\mathsf{h_K}$ that compresses the input of $\mathsf{H_K}$ to a fixed length output, as defined in the specification [SHA95]. The PRNG uses (wrongly, as we will see) the compression function $\mathsf{h_K}$ instead of $\mathsf{H_K}$ for hash calculation. When the collected input fills a block of $\mathsf{h_K}$ (of size 64 bytes), the last 20 bytes of S_1 are filled with $\mathsf{h_K}$, and then the block is set to 0 and filled again. For clarity, we denote s the current collected input and h the current calculated hash in S_1 and $I^* = [I]_0^{64-|S||I|}$ in Algorithm 3.

The next Algorithm . This algorithm is described in Algorithm 4 and in Figure 3.5. It is implemented with `engineNext Bytes`. It takes as input an integer n and outputs R, of size n bytes and the updated internal state S'. Twenty successive bytes outputs are generated as follows: the algorithm appends S_1 and S_2, calculates the output with function $\mathsf{h_K}$ (the compression function) and increments the counter contained in S_2. For clarity, we suppose that n is a multiple of 20 (the implementation allows any value with intermediate arrays whose description would complicate the understanding of the algorithm) and we denote c the counter contained in S_2. We also use the same notation (s and h) used for the refresh algorithm.

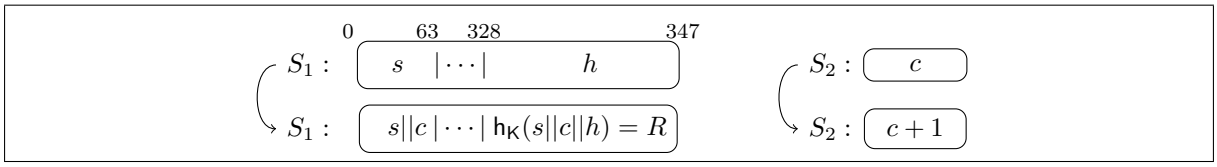


Figure 3.5: Android PRNG next algorithm

Algorithm 4 Android `SHA1PRNG` next

Require: $S = (S_1, S_2) = ([s, \cdots, h], [c]), n(n \bmod 20 = 0)$
Ensure: S', R
  for $i = 0$ to $n - 1$ **do**
    $S_1[0, \cdots, 63] = [s||c]$, $S_1[328, \cdots, 347] = \mathsf{h_K}(s||c||h)$
    $c = c + 1$
    $S_2 = [c]$
    $R_i = S_1[328, \cdots, 347]$
    $i = i + 20$
  end for
  return $S' = (S_1, S_2), R = \bigcup_i \{R_i\}$

Attack. We mount an attack against the Android `SHA1PRNG` taking in consideration the internal state decomposition. Our attack is possible because of the use of the compression function $\mathsf{h_K}$ instead of the hash function $\mathsf{H_K}$, both in the refresh and next algorithms. When using the compression function $\mathsf{h_K}$, the *current* hash value is used whereas the hash should

be calculated with the initialization vector defined in the specification [SHA95]. Again, this attack identifies a design flaw of the PRNG. This attack shows that the PRNG is not *resilient* because the attacker only needs to refresh the PRNG with an input that forces S_1 to be equal to [0]. In addition, if at initialization the internal state is filled with 64 random bytes, the PRNG is not *pseudo-random*, because no refresh is needed to mount the attack. The attack is demonstrated in Theorem 3.8 and illustrated in Figure 3.6.

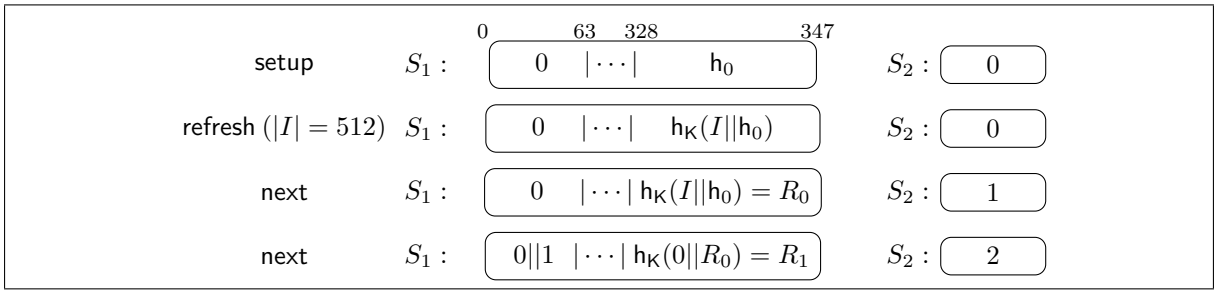


Figure 3.6: Android PRNG attack

Theorem 3.8. *Android* `SHA1PRNG` *is not resilient.*

Proof. Consider an adversary $\mathcal{A}$ against the security of the PRNG that chooses the following (one state) distribution $\mathcal{D}$, $\mathcal{D}(0) = I$, where I is of size ℓ, where $\ell \leq 512$ and random (*i.e.* $\mathcal{D}$ is legitimate with $\gamma_0 = \ell$). Next $\mathcal{A}$ makes the following oracle queries in the security game MRES: one $\mathcal{D}$-refresh, one first next-ror with an output R_1 of size 20 bytes, and one second next-ror, with an output R_2 of size 20 bytes. Then:

- After $\mathcal{D}$-refresh with I: $[S_1]_0^{63} = 0$ with probability 1/64, $[S_1]_{328}^{347}$ is random, $S_2 = 0$.
- After next-ror with R_0, $[S_1]_0^{63} = 0$ with probability 1/64, $R_0 = [S_1]_{328}^{347}$ and $S_2 = 1$.
- After next-ror with R_1, $[S_1]_0^{63} = [0||1]$, $R_1 = [S_1]_{328}^{347}$, but $[S_1]_{328}^{347} = \mathsf{h}_K(0||R_0)$ with probability 1/64.

In this last next-ror-oracle query, $\mathcal{A}$ obtains a 20-byte string that is known to $\mathcal{A}$ with probability 1/64 as it only relies on the previous output, whereas ideally, this event should occur only with probability 2^{-80}. Therefore this PRNG is not resilient. □

3.4.3 Analysis of OpenJDK `SHA1PRNG`

The OpenJDK provider contains an implementation named `SHA1PRNG`, directly given in the class `Secure Random`. This implementation follows the specification given in the Digital Signature Standard [DSS00]. This last specification has been analyzed in [KSWH98] and in [DHY02], where the authors show that it does not correspond to a *resilient* PRNG. Here we present new attacks that are based on partial corruption of the internal state.

Internal State Decomposition. The internal state of the PRNG is implemented with three private fields, the field `state`, of size 20 bytes, the field `remainder`, of size 20 bytes and an integer `remCount`. Hence the decomposition of the internal state is $S = (S_1, S_2, S_3)$, where S_1, S_2, S_3 stand for `state`, `remainder`, `remCount`, respectively and the total size of the internal state is 352 bits. The PRNG uses this decomposition as follows: S_1 contains the collected entropy, S_2 contains random bytes before their output and S_3 is used to check if S_2 contains enough random bytes that can serve as output.

The refresh Algorithm. This algorithm is described in Algorithm 5 and implemented with the method `engineSetSeed`. It takes as input the current internal state $S = (S_1, S_2, S_3)$, a new input I and outputs the new internal state by mixing S_1 with I using $\mathsf{H_K}$.

Algorithm 5 OpenJDK `SHA1PRNG` refresh

Require: $S = (S_1, S_2, S_3), I$
Ensure: S'
  $S_1 = \mathsf{H_K}(S_1||I)$
  return $S' = (S_1, S_2, S_3)$

The next Algorithm. This algorithm is described in Algorithms 6 and 7. It is implemented with two methods; the first one, `engineNextBytes`, generates the output and the second one, `updateState`, updates the internal state.

The method `engineNextBytes` takes as input the current internal state $S = (S_1, S_2, S_3)$ and n, the number of bytes requested. It outputs an n-byte output R and updates the internal state. The internal counter S_3 controls the update of the internal state when output is generated: if $S_3 > 0$, S_2 contains some bytes that have not been used for a previous output; these bytes can be used for the current output and are then set to 0. Next, S_2 and S_1 are updated only if all bytes from S_2 have been used: at first S_2 is updated with S_1 ($S_2 = \mathsf{H_K}(S_1)$) and finally S_1 is updated using `updateState` instruction, which is the implementation of the update algorithm specified in [DSS00]. The instruction `updateState` takes as input two binary strings S_1 and S_2 of size 20 bytes and mixes them together byte by byte.

Algorithm 6 OpenJDK `SHA1PRNG` next (`engineNextBytes`)

Require: $S = (S_1, S_2, S_3), n$
Ensure: S', R
  $i = t = 0$
  if $S_3 > 0$ **then**
    $t = \min\{n - i, 20 - S_3\}$
    $R[0, \cdots, t-1] = S_2[S_3, \cdots, S_3 + t - 1]$
    $S_2[S_3, \cdots, S_3 + t - 1] = [0]$
  end if
  while $i < n - 1$ **do**
    $S_2 = \mathsf{H_K}(S_1)$
    $S_1 = \texttt{updateState}(S_1, S_2)$
    $t = \min\{n - i, 20\}$
    $R[i, \cdots, i + t - 1] = S_2[0, \cdots, t - 1]$
    $i \leftarrow i + t$
  end while
  $S_3 = (S_3 + n) \bmod 20$
  return $S' = (S_1, S_2, S_3), R$

Algorithm 7 OpenJDK `SHA1PRNG` next (`updateState`)

Require: S_1, S_2, $|S_1| = |S_2| = 160$
Ensure: S_1
  $\ell = 1$
  for $i = 0$ to 19 **do**
    $v = (S_1[i] + S_2[i] + \ell)$
    $S_1[i] = v \bmod 2^8$
    $\ell = v/2^8$
  end for
  return S_1

Attack. We mount an attack against the OpenJDK `SHA1- PRNG` taking in consideration the internal state decomposition. Our attack uses the fact that S_2 and S_3 are not updated during refresh. After a refresh, if S_3 is set by the attacker to 1, the next output will be derived from a predictable value.

Theorem 3.9. *OpenJDK* `SHA1PRNG` *is not backward secure. To mount an attack against the PRNG, $\mathcal{A}$ needs to corrupt 4 bytes of the internal state.*

Proof. Consider an adversary $\mathcal{A}$ against the security of the OpenJDK `SHA1PRNG` that chooses the distribution $\mathcal{D}$, such that $\mathcal{D}(0) = I$ where I is of size 20 bytes and random (*i.e.* $\mathcal{D}$ is legitimate with $\gamma_0 = 160$). Next $\mathcal{A}$ makes the following oracle queries in the security game MBWD: one $\mathcal{D}$-refresh, one $\mathcal{M}$-set-state with $M = (0, 0, 1), J = \{3\}$ and one final next-ror with an output R of size 10 bytes. Then:

- After $\mathcal{D}$-refresh with I, $S_1 = \mathsf{H}_\mathsf{K}(I||0)$, $S_2 = 0$ and $S_3 = 0$.
- After one $\mathcal{M}$-set-state with $M = (0, 0, 1)$, $J = \{3\}$, $S_1 = \mathsf{H}_\mathsf{K}(I||0)$, $S_2 = 0$ and $S_3 = 1$.
- After one next-ror with $n = 10$, $S_1 = \mathsf{H}_\mathsf{K}(I||0)$, $S_2 = 0$, $S_3 = 11$ and $R = 0$.

Therefore, $\mathcal{A}$ obtains a 10-byte string in the last next-ror-oracle query that is predictable whereas this event should occur with probability 2^{-80}. Therefore this PRNG is not backward secure for $\gamma^* \leq 160$. Note that as the fields S_2 and S_3 are not updated during the refresh Algorithm, $\mathcal{A}$ could make sufficient calls to $\mathcal{D}$-refresh to mount a similar attack for a larger value of γ^*. □

A similar analysis can be made for the OpenJDK `Native-PRNG`. In OpenJDK, a second implementation of a PRNG is included, named `NativePRNG`, which mixes the output of the OpenJDK `SHA1PRNG` with the output of the system PRNG `/dev/urandom` using a xor instruction. Hence the internal state of the OpenJDK `NativePRNG` is the concatenation of the internal state of the PRNG `/dev/urandom` with the internal state of the OpenJDK `SHA1PRNG`. The analysis of `/dev/urandom` done in [DPR+13] gives the details about the PRNG `/dev/urandom` and its internal state decomposition, and we obtain directly that OpenJDK `NativePRNG` has an internal state of size 5472 bits. Following their analysis, we can show that OpenJDK `NativePRNG` is not robust, and that an attacker needs to corrupt 128 bytes of `/dev/urandom` and 4 bytes of `SHA1PRNG` to mount an attack against `NativePRNG`.

3.4.4 Analysis of Bouncycastle `SHA1PRNG`

The Bouncycastle Crypto package is a Java implementation of cryptographic algorithms; our analysis refers to release 1.5 [Bou]. The implementation of several PRNGs is contained in the package `org.bouncycastle.crypto.prng`, where the implementation of the `SHA1PRNG` is in the class `DigestRandomGenerator`. The implementation combines a cryptographic hash function (which is by default H_K) with internal instructions that are used to update the internal state of the PRNG. In our source code analysis, we identified several weaknesses: first a weakness related to the decomposition of the internal state, and second a weakness due to an incomplete state update during the refresh algorithm. These weaknesses have neither been identified in [MMS13], nor by the Bouncycastle community.

Internal State Decomposition . The internal state of the PRNG is implemented with the following fields: `seed` of size 160 bits, `state` of size 160 bits, `seedCounter` of size 64 bits, and field `stateCounter`, of size 64 bits. The two first fields contain the collected entropy and the two last fields are counters that are used for PRNG operations. Hence, the total size of the internal state is 448 bits and its decomposition is $S = (S_1, S_2, S_3, S_4)$, where S_1, S_2, S_3, S_4 stand for `seed`, `state`, `seedCounter`, `stateCounter`, respectively.

The refresh Algorithm. This algorithm is described in Algorithm 8. It takes as input the current internal state (S_1, S_2, S_3, S_4) and an input I; it outputs a new internal state where only S_1 is updated. It is implemented with the method `addSeedMaterial`.

Algorithm 8 Bouncycastle `SHA1PRNG` refresh

Require: $S = (S_1, S_2, S_3, S_4)$, I
Ensure: S'
$S_1 = \mathsf{H}_\mathsf{K}(S_1||I)$
return $S' = (S_1, S_2, S_3, S_4)$

The next Algorithm. This algorithm is described in Algorithms 9 and 10. It is implemented with the method `NextBytes`. It takes as input an integer n, the current the internal state (S_1, S_2, S_3, S_4) and outputs an n-byte string R. The output R is derived from S_2, while an internal method, named `generateState` is used to update the state.

The `generateState` method increments the counters S_3 and S_4 and calculates the new values of S_1 and S_2 accordingly.

Attack . We mount an attack against the Bouncycastle `SHA1PRNG` taking into consideration the internal state decomposition. This attack is similar as the attack against [DSS00] described in [KSWH98] and [DHY02]: the attacker uses a previously generated ouput as an input to corrupt the PRNG: our attack shows that Bouncycastle `SHA1PRNG` is not *resilient*.

Theorem 3.10. *Bouncycastle* `SHA1PRNG` *is not resilient.*

Proof. Consider an adversary $\mathcal{A}$ against the resilience of the PRNG that chooses the following (2-state) distribution $\mathcal{D}$, $\mathcal{D}(0) = I, \mathcal{D}(1) = J$, where I and J are of size 20 bytes, I is random and J is known by $\mathcal{A}$ (*i.e.* $\mathcal{D}$ is legitimate with $\gamma_0 = \gamma_1 = 160$). Next $\mathcal{A}$ makes

Algorithm 9 Bouncycastle `SHA1PRNG` `next` (`NextBytes`)

Require: $S = (S_1, S_2, S_3, S_4)$, n
Ensure: S'
  $S = \texttt{generateState}(S)$
  $j = n$
  for $i = 0$ to j **do**
    if j $= 20$ **then**
      $S = \texttt{generateState}(S)$
      $j = 0$
    end if
    $R[i] = S_2[i]$
    $i = i + 1$
  end for
  return $S' = (S_1, S_2, S_3, S_4)$, R

Algorithm 10 Bouncycastle `SHA1PRNG` `next` (`generateState`)

Require: $S = (S_1, S_2, S_3, S_4)$
Ensure: S'
  $S_4 = S_4 + 1$
  $S_2 = \mathsf{H}_\mathsf{K}(S_4||S_2||S_1)$
  if $S_3 \bmod 10 = 0$ **then**
    $S_3 = S_3 + 1$
    $S_1 = \mathsf{H}_\mathsf{K}(S_1||S_3)$
  end if
  return $S' = (S_1, S_2, S_3, S_4)$

the following oracle queries in the security game MRES: one $\mathcal{D}$-refresh, two next-ror with two outputs R_1 and R_2, both of size 20 bytes, one $\mathcal{D}$-refresh, and one third next-ror, with one output R_3 of size 20 bytes. Then:

- After one $\mathcal{D}$-refresh with I, $S_1 = \mathsf{H}_\mathsf{K}(I||0)$, $S_2 = 0$, $S_3 = 1$, $S_4 = 1$.
- After one next-ror, with $|R_1| = 20$, S_1 remains the same, $S_2 = \mathsf{H}_\mathsf{K}(S_4||S_2||S_1) = \mathsf{H}_\mathsf{K}(2||0||S_1)$, $S_3 = 1$, $S_4 = 2$, $R_1 = S_2$.
- After one second next-ror, with $|R_2| = 20$, S_1 stays the same, $S_2 = \mathsf{H}_\mathsf{K}(S_4||S_2||S_1) = \mathsf{H}_\mathsf{K}(3||R_1||S_1)$, $R_2 = S_2$.
- After one $\mathcal{D}$-refresh with $J = [3||R_1]$, $S_1 = \mathsf{H}_\mathsf{K}(J||S_1) = \mathsf{H}_\mathsf{K}(3||R_1||S_1) = R_2$.
- After one last next-ror with $|R_3| = 20$, S_1 remains the same, $S_2 = \mathsf{H}_\mathsf{K}(S_4||S_2||S_1) = \mathsf{H}_\mathsf{K}(4||R_2||R_2)$, $R_3 = S_2$.

Therefore, $\mathcal{A}$ obtains a 20-byte string in the last next-ror-oracle that is predictable ($R_3 = \mathsf{H}_\mathsf{K}(4||R_2||R_2)$), whereas this event should occur with probability 2^{-80}. Therefore the Bouncycastle `SHA1PRNG` is not resilient. □

3.4.5 Analysis of IBM `SHA1PRNG`

Besides Oracle's Java Virtual Machine, IBM implements its own JVM with some differences (in particular in performance) compared to Oracle's JVM. We analyze the IBM SDK Version 7 Service Refresh 7 which contains a security enhancement of the PRNG reported by Sethi in [IBM14]. We analyze the implementation of the crypto provider `IBM-SecureRandom`, in the package `com.ibm.securerandom.pro-vider`. This (closed source) implementation consists of a main entropy pool and a mixing function which internally relies on the hash function family H_K to update the pool.

Internal State Decomposition. The internal state of the IBM `SHA1PRNG` is self-contained in the field `state` of size 680 bits. For convenience, we refer to the field `state` as $S = (S_1||S_2||S_3||S_4||S_5||S_6||S_7)$. The IBM `SHA1PRNG` uses this decomposition as follows: S_1 contains the number of bytes that has been used from the output pool, $S_2 = 0$, S_3 is the output, S_4 is a first entropy pool, S_5 are 5 different internal counters, S_6 is a second entropy pool and S_7 is a flag indicating whether the input is provided or not. The initial state is $S_1 = 0, S_2 = 0, S_3 = 0, S_4 = 0, S_5[0] = 0, S_5[1] = 128, S_5[2] = 30, S_5[3] = 0, S_5[4] = 0, S_6 = 0, S_7 =$ `false` and it relies on the internal function `reverse` that simply reverses binary the content of the input.

The refresh algorithm. This algorithm is described in Algorithm 11. It takes as input the current internal state $(S_1, S_2, S_3, S_4, S_5, S_6, S_7)$, a input I and outputs the new internal state by mixing S_4 with I using H_K. It is implemented with the method `engineSetSeed`.

Algorithm 11 IBM `SHA1PRNG` refresh

Require: $S = (S_1, S_2, S_3, S_4, S_5, S_6, S_7), I$
Ensure: S'
  if $|I| > 320$ **then**
    $S_6 = \mathsf{H}_\mathsf{K}(I)$
  end if
  $\bar{I} =$ `reverse(I)`
  $S_4 = S_4 \oplus \bar{I}$
  $S_7 =$ `true`
  $S_1 = |S_3|$
  return $S' = (S_1, S_2, S_3, S_4, S_5, S_6, S_7)$

The next algorithm. This algorithm is described in Algorithms 12 and 13. It is implemented with the methods `engineNextBytes` and `updateEntropyPool`. It takes as input the current internal state S and n, the number of bytes requested. It outputs an n-byte R and a new value for the internal state. It relies on S_1 to generate the output as follows: if $S_1 < |S_4|$, S_3 still contains bytes that have not been used in a previous output. When S_1 reaches the size of the entropy pool (i.e. $S_1 = |S_4|$), S_3 and S_4 are updated to produce a fresh output. First entropy is added by the internal method `updateEntropyPool` and then the output pool $S_3 = \mathsf{H}_\mathsf{K}(S_3||S_4||S_5||S_6[1])$ is updated. The instruction `time` returns the timestamp, δ is another timestamp value, and `firstTime` is an internal flag in order to ensure that S_3 is indeed filled. This procedure is repeated for each $|S_3|$ bytes.[1]

[1] In practice, the size of the output pool is 20 bytes.

Algorithm 12 IBM `SHA1PRNG` next (`engineNextBytes`)

Require: $S = (S_1, S_2, S_3, S_4, S_5, S_6, S_7), n$
Ensure: S', R
  if `firstTime` $=$ `true` **then**
    if $S_1 = |S_3|$ **then**
      $(S_4, S_5, S_7) =$ `updateEntropyPool`(S)
    end if
    $S_3 = \mathsf{H}_\mathsf{K}(S_3||S_4||S_5[0]||S_5[1])$
    $S_1 = 0$
  end if
  $R = S_3[S_1, \ldots, n]$
  $S_1 = 1 + n$
  return $S' = (S_1, S_2, S_3, S_4, S_5, S_6, S_7), R$

Algorithm 13 IBM `SHA1PRNG` next (`updateEntropyPool`)

Require: $S = (S_1, S_2, S_3, S_4, S_5, S_6, S_7), I$
Ensure: S_4, S_5, S_7
  if $S_5[1] > 0$ && $S_7 =$ `false` **then**
    for $S_5[0]$ to $S_5[0] + 20$ **do**
      if `time` $\geq S_5[4] + S_5[5]$ **then**
        $S_4 = S_4 \oplus I$
        $S_5[4] = \delta$
        $S_5[5] + S_5[2] +$ `time`
        $S_5[0] + 1$
      end if
    end for
  end if
  return (S_4, S_5, S_7)

Attack . We mount an attack similar to the attack on the OpenJDK `SHA1PRNG`. As in the refresh algorithm the internal state is not completely updated, an attacker can set the byte $S_1 = 0$ and make the counter of non-used bytes start reading again from $S_3[0]$. Notice that we need at least 3 bytes to set $S_1, S_5[4], S_5[5]$ properly otherwise the algorithm will force to add entropy; on the other hand, once all parameters are set up, an attacker just needs to corrupt 1 integer (4 bytes) to make the output predictable.

Theorem 3.11. *IBM* `SHA1PRNG` *is not backward secure. To mount an attack against the PRNG, $\mathcal{A}$ needs to corrupt 4 bytes of the internal state.*

Proof. Consider an adversary $\mathcal{A}$ against the security of IBM `SHA1PRNG` that chooses a distribution $\mathcal{D}$, such that $\mathcal{D}(0) = I$ where I is of size 20 bytes and random (*i.e.* $\mathcal{D}$ is legitimate with $\gamma_0 = 160$). Next $\mathcal{A}$ makes the following oracle queries in the security game MBWD: one $\mathcal{D}$-refresh, one next-ror with an output of size 10 bytes, one $\mathcal{M}$-set-state with $M = (0, 0, 0, 0, 0, 0, 0), J = \{3\}$ and one final next-ror with an output of size 10 bytes. Then:

- After one $\mathcal{D}$-refresh with I, $S_1 = |S_3|, S_2 = 0, S_3 = 0, S_4 = 0 \oplus I, S_5[0] = 0, S_5[1] = 128, S_5[2] = 30, S_5[3] = 0, S_5[4] = 0, S_6 = 0, S_7 =$ `true`.

- After one next-ror with $n = 10$, $S_1 = 10, S_2 = 0, S_3 = \mathsf{H}_\mathsf{K}(0||0 \oplus I||0||128), S_4 = 0 \oplus I, S_5[0] = 0, S_5[1] = 128, S_5[2] = 30, S_5[3] = 0, S_5[4] = 0, S_6 = 0, S_7 = \texttt{true}$. $R = S_3[0, \ldots, 10]$. The output R is random.
- After one $\mathcal{M}$-set-state with $M = (0, 0, 0, 0, 0, 0, 0)$, $J = \{1\}$, $S_1 = 1, S_2 = 0, S_3 = \mathsf{H}_\mathsf{K}(0||0 \oplus I||0||128), S_4 = 0 \oplus I, S_5[0] = 0, S_5[1] = 128, S_5[2] = 30, S_5[3] = 0, S_5[4] = 0, S_6 = 0, S_7 = \texttt{true}$.
- After one next-ror with $n = 10$, $S_1 = 10, S_2 = 0, S_3 = \mathsf{H}_\mathsf{K}(0||0 \oplus I||0||128), S_4 = 0 \oplus I, S_5[0] = 0, S_5[1] = 128, S_5[2] = 30, S_5[3] = 0, S_5[4] = 0, S_6 = 0, S_7 = \texttt{true}$ and $R = S_3[0, \ldots, 10]$.

Therefore, $\mathcal{A}$ obtains a 10-byte string in the last next-ror-oracle query that is exactly the same as the previous next-ror-oracle query, whereas ideally, this event occurs only with probability 2^{-80}. Therefore the IBM SHA1PRNG is not backward secure for $\gamma^* \leq 160$. Note that as the fields S_2 and S_3 are not updated during the refresh Algorithm, $\mathcal{A}$ could make sufficient calls to $\mathcal{D}$-refresh to mount a similar attack for a larger value of γ^*. □

3.4.6 Towards a Secure Implementation

In [DPR+13], Dodis *et al.* proposed a construction based on simple operations in a finite field. Let $\mathbf{G} : \{0,1\}^m \to \{0,1\}^{n+\ell}$ be a (deterministic) pseudorandom generator where $m < n$. The PRNG $\mathcal{G}$ is defined as follows:

- setup(): output seed $= (X, X') \xleftarrow{\$} \{0,1\}^{2n}$.
- $S' =$ refresh(S, I): Given seed $= (X, X')$, current state $S \in \{0,1\}^n$, and a sample $I \in \{0,1\}^n$, output: $S' := S \cdot X + I$, where all operations are over $\mathbb{F}_{2^n}$.
- $(S', R) =$ next(S): Given seed $= (X, X')$ and a state $S \in \{0,1\}^n$, first compute $U = [X' \cdot S]_1^m$. Then output $(S', R) = \mathbf{G}(U)$.

In [DPR+13], Dodis *et al.* proved the following theorem:

Theorem 3.12. *Let k, m, ℓ, n be integers, where $n \geq m + 9k + 1$ and $\gamma^* = m + 8k + 1$. Let $\mathcal{G}$ be defined as above. Then $\mathcal{G}$ is a $((t', 2^k, 2^k, 2^k), \gamma^*, 2^{k+1} \cdot \varepsilon + 2^{-k})$-robust PRNG, where $t' \approx t$.*

Theorem 3.12 shows that the PRNG $\mathcal{G}$ resists a total internal state corruption. In [DPR+13], Dodis *et al.* give concrete values for a concrete instanciation of the PRNG $\mathcal{G}$; Namely, they obtain, with $\mathbf{G}(U) = (\mathsf{AES}_U(0), \ldots, \mathsf{AES}_U(6))$, $n = 705$, $m = \ell = 128$ and $\gamma^* = 641$. As our analysis shows, the implementation of the PRNG (and especially the way the internal state is updated during PRNG operations) may be used by an attacker to corrupt the PRNG. Then starting from the definition of the PRNG $\mathcal{G}$ and its concrete instanciation given with $\mathbf{G}$, n, m and ℓ above, one can implement a secure PRNG provided the internal state decomposition and its update during PRNG operation are analyzed with care to ensure that the implementation does not contain any vulnerability.

3.5 Memory Corruption

Partial memory corruption in some cases may be easier than corruption of the entire memory. For example, by excessively incrementing or decrementing an array pointer in a loop without proper bound checking a buffer overflow may happen, which can be exploited to read or overwrite sensitive data. Using this technique it is possible to get a partial or

```
JAVA_OPTIONS='-Xdebug -Xrunjdwp:transport=dt_socket,
address=8998,server=y,suspend=n'
```

Figure 3.7: Modification of the Java Virtual Machine

total information from the memory as the Heartbleed bug [Hea14]. A complete survey of memory attacks has been written by Szekeres *et al.* in [SPWS13].

3.5.1 Proof of Concept

As a proof of concept, we describe the technical details of a malicious program that partially sets the memory of a Java PRNG. In a first stage we present how to interact with the Java Virtual Machine and in a second stage we present how to interact with the PRNG.

The Java Virtual Machine. Our work is based on Java execution model, particularly the Java 7 update 51. Java source code is compiled into Java Virtual Machine instructions (or bytecodes) and is executed in a abstract computing machine called Java Virtual Machine (JVM). The JVM translates the bytecode into specific machine code instructions and manages the memory for Java applications. One way to establish a connection between an application running inside the Virtual Machine and an external application is the Java Platform Debugger Architecture (JPDA) which is a set of protocols and interfaces that provide a standardized infrastructure for third-party debuggers. The JDPA is fully described in [JDP]: it defines a set of instructions to control the application execution and memory management. The JPDA defines a communication protocol which is called the Java Debug Wire Protocol (JDWP). Using this communication protocol, it is possible to debug a running application remotely or locally, modify local variables, etc. To use this, the Java options must be enabled in the operating system environment as described in Figure 3.7. Hence using standard instructions that are defined in [JDP], the malicious program can simulate a debugger, get access to all memory fields used by a Java application and set them to chosen values.

Internal State Modification using the Debugging Facilities. All Java implementations studied in this work use one or more private field(s) that are available for modification with a debugger connected to the Java application using a socket. Once connected, the malicious program interacts with the application with the standard instructions. Precisely, it only needs to use instruction `stop at` to stop the execution of the application and instruction `set` to set a variable used by the application to a chosen value. These instructions are described in Table 3.1.

Table 3.1: Internal State Modification

Java Debug Interface	Comments
stop at *Class:line*	Stop execution at line
set *variable* = *value*	Change local variable value

Attack Description. First decompile the Java bytecode to convert it in source code. The decompiled binary can be attached to a debugger process in order to ease code inspection, examine variables and watch control flow. Using this, we propose some simple stealthy malware:

1. Select an application to attack.
2. Decompile it, extract the source code and check whether is susceptible to be attacked.
3. Export the JVM options to enable remote debugging.
4. Attach the source code generated to the debugger and add breakpoints.
5. Modify variables and internal states after breakpoints are triggered.
6. Continue with application execution.

3.5.2 An Attack against a Java Tor Client.

To illustrate our attack, we use the previously described malware[2] to compromise the PRNG of a pure Java Tor Client named Orchid. This Tor client is implemented using the specifications of the Tor protocol, as described in [Tor] and in [Orc]. First, we downloaded the JAR file from developer's website [Orc]. As usual, the JAR (Java ARchive) file is a container of the bytecode, resources and metadata. We decompiled the archive using the Java decompiler available from [JD] and we analyzed the extracted source code to identify the PRNG used, where we could set the breakpoints in order to stop the program execution and which fields to compromise. With all this information, the malware is crafted and is delivered to the computer's victim. The first task of the malware is to modify the JVM options, adding the command option describes in Figure 3.7 to the environment variables in Linux/Unix or modifying the Windows Registry in Windows. When adding this entry to the environment, the JVM is launched listening to the port selected (in this case, 8998) for a remote connection. The malware can then connect from the same computer via sockets, hence it does not need to send information to outside, preventing detection from networking monitors. Using the source code extracted from the binary code, we deduced that the Tor client Orchid instances `SHA1PRNG` from provider SUN (equal to `SHA1PRNG` from provider OpenJDK deduced in Section 3.4.3) as PRNG inside the `TorRandom` class. Then, as described in Figure 3.7 the high-level malware connects to port 8998. When the debugger connects to the application, it waits for the `nextInt()` method call for obtain a new random Integer. This method calls internally `engineNextBytes()`, which is the method for random generation and is the target of our attack. As we showed in Section 3.4.3, the critical field of the internal state of the PRNG to modify is the integer `remCount`. Once the application has stopped, the malware inspects the internal state of the PRNG, modifies the identified critical field in the PRNG and continues with the execution of the application without user awareness. As explained, no remote interaction is require to perform this attack so these operations are only local. In the context of the Tor network, the client takes a random pathway to route the data packets and randomness is required to ensure privacy of the client on the network. The Tor client Orchid generates this pathway with the method named `getRandomlyOrderedListOfExitCircuits`. It makes a call to `nextInt()`. Compromising the PRNG by making the output predictable, we can force to select always one particular path (probably compromised) for all data communications of the node.

[2] Also can be performed using a 0-day Java exploit or similar.

3.6 Conclusion

We proposed a new security model for PRNG analysis, where an attacker has partial access to the internal state and we model the expected properties of the PRNG. This new security model is based on the most recent and strongest security model called robustness of PRNG and it is closely related to its real-life use and implementation. It states that the PRNG should continue to generate non predictable outputs even if its internal state is partially corrupted, and models real-life situations, where a PRNG environments may be adversarial and running applications can be partially corrupted.

We analyzed several widely used PRNG (from providers OpenSSL, OpenJDK, Android, IBM and Bouncycastle) by clearly describing their operations in the security model. In particular, we showed that all of them are highly sensitive to a relatively small corruption of their internal state. This vulnerability is due to the concrete implementation of their internal state that relies on several fields between which transfers are done, controlled by internal values that can be set by the attacker. Moreover, we showed that for two providers (Android and Bouncycastle), internal state corruption is not required to break the PRNG. This work shows that proper implementation of PRNG requires a lot of attention and should therefore rely on proven constructions.

Chapter 4

Using the Cloud: Key Management

4.1 Introduction

Nowadays, cloud storage is quite popular with zettabytes of data spread all over the world. Even if providers give some backup guarantees, they cannot always prevent compromises, and so the data are subject to leakage, with possibly huge consequences if the data are sensitive (financial, economic, medical, etc). Clearly, the provider can encrypt the data before storing them, but this is not an end-to-end protection for the user: the provider itself has access to the data. For better security, the user should encrypt the data before sending them to the cloud. But this leads to a key management issue: Users have to remember their secret keys!

Humans cannot remember large secret keys, but just low-entropy passwords (and not too many). Such a password is definitely not enough to deterministically derive a symmetric encryption key, since a simple offline dictionary attack would allow the recovery. On the other hand, there are techniques using passwords that are not vulnerable to such offline dictionary attacks, like *password authenticated key exchange* (PAKE) [BM92]. For these PAKE protocols, the best attacks require the adversary to be online, and to make the exhaustive search by interacting with the honest parties, hence the idea to combine PAKE with secret sharing, in order to achieve the best of the two worlds. This allows the recovery of a high-entropy symmetric key by interacting with several servers while just using a low-entropy password [FK00, Jab01], without relying on any authenticated data, where the best attacks are online dictionary attacks.

Password-Protected Secret Sharing. A (t, n) *Password-Protected Secret Sharing* (PPSS) is a protocol that allows a user to reconstruct a high-entropy secret from a single (human-memorable) password, by communicating with at least $t+1$ honest servers (among n possible ones).

This framework formalized in [BJSL11] first defines a secure *initialization* phase where the secret is processed together with the password, and some server information, in order to distribute the secret among n independent servers. Only public information (to enable the later reconstruction) is eventually stored on each server. We however stress that this public information does not have to be authentic for the later security. Then, during the *reconstruction* phase, the user can recover his secret by interacting with any subset of $t+1$ honest servers using just his password. If the public information has been altered, the knowledge of the password will be enough to detect it. However, in [BJSL11] they prove

their scheme secure in the random oracle model assuming an additional PKI. Whereas this assumption of a safe PKI makes sense during the initialization phase, which can be run in a safe environment, it is not reasonable to make this assumption for the reconstruction phase, which will be executed many times on various weak devices.

A PPSS protocol satisfies the following properties: (i) the user can retrieve the data by executing the reconstruction protocol with the same password as the one used in the initialization phase and it is guaranteed to succeed as long as at least $t+1$ honest servers are available. (ii) An attacker who controls up to t servers cannot learn any information about the secret other than doing an online dictionary attack with another server. Two additional properties have been defined: *Soundness* and *Robustness*. The first guarantees that even if the adversary compromises all the servers, it cannot make the user reconstruct a secret different from the one originally stored by the user. On the other hand, robustness guarantees the recovery of the secret as long as the user communicates without disruptions with at least $t+1$ honest servers.

Additionally, we point out that the adversary can control all the communication network by blocking, delaying, altering, or duplicating any flow. As such, no server is trusted, and no PKI is assumed either, since the only authenticated data we allow is a short password that the user can remember.

4.1.1 Related Work.

A *threshold secret sharing scheme* allows a user to distribute a secret among different participants preventing a sole party breaking the security or obstructing the reconstruction. This idea was introduced by Shamir [Sha79] and Blakey [Bla79]. This concept was later generalized by using two thresholds, a *upper* and a *lower* one to set the size of the sets to reconstruct and to preserve privacy respectively. In Shamir's secret sharing scheme, the privacy threshold is defined as t and the reconstruction threshold as $t+1$. When this gap is higher, then the secret sharing scheme is called *ramp* scheme [BM84, BSV93, MJ96, MS81]. Ramp schemes to achieve a robust secret sharing scheme have been extensively studied, we refer the reader to [Che15] and [BPRW15].

The first formal definition of Password Protected Secret Sharing was introduced by Bagherzandi *et al.* [BJSL11]. They proved their scheme secure in the random oracle model assuming an additional PKI. Moreover, if an adversary is able to obtain the keypair of one server, the adversary can perform an offline attack. Later, Camenisch *et al.* [CLN12] introduce a protocol of password-authenticated secret sharing that also assumes a PKI and only two servers. Both protocols contradict the requirement to be *password-only*, since they assume additional authenticated data. Whereas this assumption of a safe PKI makes sense during the initialization phase, which can be run in a safe environment, it is not reasonable to make this assumption for the reconstruction phase, which will be executed many times on various weak devices.

Later, Camenisch *et al.* [CLLN14] introduce a (t, n)-PPSS (called TPASS, for Threshold Password-Authenticated Secret Sharing) in the Universal Composability (UC) framework [Can01] that is password-only during the reconstruction phase. However, in this protocol all servers jointly validate if the password matches or not. Yi *et al.* in [YHCL15] propose a more efficient TPASS based on distributing the password, a secret and a digest of the secret. Nevertheless, in the recovering protocol, at least t servers execute a broadcasting protocol to generate and return the ElGamal encryptions of both the secret and the digest. Then the users verify it matches.

Camenisch *et al.* in [CLN15] present a very lightweight protocol with a similar construction to our work, yet with differences. Each server holds a key that is refreshed at regular time intervals that allows them to recover from corruption through a non-interactive key refresh protocol making it unfeasible to perform an offline attack unless all servers are corrupted at the same time. Since this protocol does not rely on robust secret sharing scheme nor zero-knowledge, it is not possible to identify which shares are valid. Then, if in the end the validation fails, the protocol must restart with a different set of servers contradicting the requirement of *robustness* and leading to a possible Denial-of-Service (DoS) attack.

Jarecki *et al.* [JKK14] have been the first to design a PPSS scheme that is both *password-only* during the reconstruction phase and *robust*, to avoid easy DoS attacks. It makes use of a *Verifiable Oblivious Pseudorandom Function* (VOPRF) that assures robustness by providing computation guarantees from the servers: the user actually knows which server has tried to cheat, or which communication links have been altered. Recently, the work [JKKX16] improves the performance of this password-only PPSS on the cost of dropping the robustness property. Their protocol is relaxing the verifiable property of the OPRF, giving up the ability to discard incorrect computations during interactions with servers. This can be a good alternative for a small number n of servers, the only setting that allows checking in a reasonable time different subsets of servers until finding a non-corrupted one.

The solution we propose in this work follows the previous strategies: we use a secret sharing scheme to divide the user's secret. Each server stores one share masked by the pseudorandom value computed in an oblivious way on the user's low-entropy password with the server's PRF key.

4.2 Preliminaries

We review the well-known computational assumptions and the classical building blocks respectively and we present a high-level description of the PPSS protocol, to motivate the needs, with first an initialization phase and then a reconstruction phase.

4.2.1 High-Level Description

Each server S_k owns a key-pair $(\mathsf{sk}_k, \mathsf{pk}_k)$ that defines a PRF F_k, with public parameters defined by pk_k and a secret key defined by sk_k. For a password $\mathsf{pw} \in \mathcal{D}$, the user asks for an oblivious evaluation of $\pi_k = F_k(\mathsf{pw})$ to n servers, where $\Pi = (\mathsf{pk}_k)_k$ is the tuple of the public keys of the involved servers. The secret key K is then split into shares $(s_1, \ldots, s_n)$ and some extra public information PInfo, specific to the user is derived from it and distributed to all servers. This information allows the user to later recover his secret, in a robust way.

We stress that during this initialization phase, $(\mathsf{pk}_k)_k$ are all the true public keys, and $(\pi_k)_k$ are the correct evaluations of the PRFs. However, during the reconstruction phase, the values provided by the servers are sent through an insecure channel and they might be altered by the adversary: the user interacts with at least t_r servers, that provide him PInfo, and help him to compute each $\pi_k = F_k(\mathsf{pw})$ in an oblivious way. We assume that the user received the same value PInfo from at least t_r servers, and then the user keeps the majority value. Using PInfo and enough evaluations π_k, the user can extract enough shares among $(s_1, \ldots, s_n)$ and reconstruct a value K. He can then verify whether this is the expected

secret key, from the majority PInfo which is however not considered authentic. We can note that there are two crucial tools for this generic construction:

- a pseudorandom function F that can be evaluated in an oblivious way: the server input is the secret key sk and the user input is the password pw, and the user only gets the output $F_{\mathsf{sk}}(\mathsf{pw})$, but none of the players learn any additional information about the other player's input;
- a (t_ℓ, t_r, n)-threshold secret sharing scheme that allows to share a secret among n players so that any subset of t_r shares allows efficient reconstruction of the secret, while t_ℓ shares do not leak any information.

An additional non-malleable commitment scheme [DIO98] will provide the soundness, by limiting the ability for an adversary to present a modified PInfo, whereas it controls all the communications.

However, in order to achieve the robustness to the PPSS protocol, we need to make sure that when t_r communications with the servers are unmodified, the user can reconstruct the secret: either one can detect alterations of the communications during the oblivious evaluations of the PRF, which is the approach followed by [JKK14] with *Verifiable Oblivious PRFs* (VOPRFs), or one can efficiently reconstruct a secret from any set of shares that contains at least t_r valid shares, which is our approach with *Robust Gap Threshold Secret Sharing Scheme*.

4.2.2 Computational Assumptions

We consider a finite multiplicative cyclic group $\mathbb{G} = \langle g \rangle$ of prime order q.

Computational Diffie-Hellman Assumption (CDH). The CDH_g assumption states that given g^x and g^y, where x and y were drawn at random from $\mathbb{Z}_q$, it is hard to compute g^{xy}. We denote by $\mathsf{Succ}^{\mathsf{cdh}}(\mathcal{A})$ the success probability of the adversary $\mathcal{A}$ in computing g^{xy}, and more generally, $\mathsf{Succ}^{\mathsf{cdh}}(t)$ is the best success probability an adversary can get within time t.

Decisional Diffie-Hellman Assumption (DDH). The DDH_g assumption states that given one of the two tuples (g^x, g^y, g^{xy}) and (g^x, g^y, g^z) where x, y, z are chosen at random and independently from $\mathbb{Z}_q$, no efficient algorithm can distinguish between them. We denote by $\mathsf{Adv}^{\mathsf{ddh}}(\mathcal{A})$ the advantage of the adversary $\mathcal{A}$ in distinguishing between the two distributions, and more generally, $\mathsf{Adv}^{\mathsf{ddh}}(t)$ is the best advantage an adversary can get within time t.

Gap Diffie-Hellman Assumption (GDH). The GDH_g assumption [OP01] states that the CDH_g assumption holds even when the adversary has access to a DDH_g oracle that exactly answers for any query $\mathsf{DDH}_g(g^x, g^y, g^z)$ whether $z = xy$ or not. $\mathsf{Succ}^{\mathsf{gdh}}(\mathcal{A})$ and $\mathsf{Succ}^{\mathsf{gdh}}(t, q_d)$ are defined as above, where $\mathcal{A}$ can ask up to q_d DDH_g oracle queries.

One-more Gap Diffie-Hellman Assumption (OMGDH). The (n, m)-One-more Gap Diffie-Hellman assumption [BNPS03] states that given g^x where $x \xleftarrow{\$} \mathbb{Z}_q$, a list $(g_1, \cdots, g_n) \xleftarrow{\$} \mathbb{G}^n$, unlimited access to a $\mathsf{DDH}_g(g^x, \cdot, \cdot)$ oracle, and up to m queries to a $\mathsf{CDH}_g(g^x, \cdot)$ oracle, it is hard to output $m + 1$ valid pairs (g_i, g_i^x).

$\mathsf{Succ}^{\mathsf{omgdh}}(n, m, \mathcal{A})$ and $\mathsf{Succ}^{\mathsf{omgdh}}(n, m, t, q_d)$ are defined as above, where $\mathcal{A}$ can ask up to q_d DDH_g oracle queries.

4.2.3 Building Blocks

Threshold Secret Sharing Scheme. A (t, n)-threshold secret sharing scheme splits a secret s into n shares, distributed to n participants in such a way that any subset of t $(0 < t \leq n)$ participants with valid shares is able to reconstruct the original secret, whereas any subset of less than t participants leaves the secret completely undetermined.

A (t, n)-threshold secret sharing scheme is called *perfect* if any subset smaller than t has no information at all about the secret, in an information-theoretic sense. More precisely, a (t, n)-threshold secret sharing scheme is defined on a set of n participants $P_1, \ldots, P_n$, with algorithms `ShareGen`and `Reconstruct`:

- `ShareGen`(s, t): on a secret s and a threshold t, this algorithm generates n shares $(s_1, \ldots, s_n)$, and possible public information SSInfo;
- `Reconstruct`$(\{s_i\}, \mathsf{SSInfo})$: on a set of t shares, and the possible additional information SSInfo, this algorithm recovers the secret s.

The correctness guarantees that the `Reconstruct` algorithm recovers the correct initial secret on any set of t shares. Such a scheme is said secure if any set of less than t shares cannot reconstruct the secret.

The notion of the threshold secret sharing scheme has been extensively studied, and extensions like verifiability (which is the capability for the participants to verify their shares are correct), robustness, cheater detection, and cheater identification, among others, have been proposed to this basic model [MS81, TW88, Oba11, CFOR12, JS13, LP14].

Verifiable secret sharing schemes actually allow verifiability of individual shares, using the additional SSInfo that contains verifiers for every shares. In our proposal we want to have verifiability of shares at a more global level only and avoid individual verifiability because it could allow to a unique corrupted server make an off-line dictionary attack on its own. However, when a subset of valid and invalid shares is given, without verifiability, it is in general quite difficult to extract a subset of t valid shares and recover the secret. The unique solution is often the exhaustive search among all the subsets of t shares, which requires an exponential time (in n).

Robust Threshold Secret Sharing Scheme. Several notions of robustness have been defined in the literature for secret sharing schemes. For our purpose, a secret sharing scheme will be said *robust* if, when a user is given m shares with at least t_r valid shares, he can efficiently recover the secret. It will be said *robust with respect to random failures* when the reconstruction is only possible if invalid shares are random, and not fabricated by the adversary, which is enough for our purpose.

In the following, we present a generic technique, to enhance a (t, n)-threshold secret sharing scheme, that allows to efficiently find the appropriate subset of t valid shares among a set of candidates, without increasing the size of the shares. More precisely, we will assume that we have a set of m candidates, with at least t correct values, whereas the incorrect values are random. To this aim, the additional public information SSInfo will contain global information on the shares only, and no information on the individual shares: for the construction we propose in this work, SSInfo is the product of all the fingerprints, modulo a small prime, in order not to leak too much information.

Oblivious Pseudorandom Functions. A pseudorandom function [GGM86] (PRF) is actually a keyed-family of functions $(F_k)_k$, where the outputs are indistinguishable, for a random key k, from random elements in the function range. An *oblivious* PRF (OPRF) [FIPR05] is a protocol that allows the sender contribute the key k and the receiver compute the value of $F_k(x)$ on any input of x of the receiver in a way that the sender learns nothing from the protocol.

Encryption Schemes. A public-key encryption scheme is a triple $(\mathcal{K}, \mathcal{E}, \mathcal{D})$ of algorithms. The key generation algorithm $\mathcal{K}$ takes as input a security parameter and outputs an encryption/decryption key pair $(\mathsf{ek}, \mathsf{dk})$. The encryption algorithm $\mathcal{E}$ takes as input an encryption key ek and a message m and outputs a ciphertext c. The decryption algorithm $\mathcal{D}$ takes as input a decryption key dk and a ciphertext c and outputs either the decryption m of c or $\perp$. The correctness condition required is that for all $(\mathsf{ek}, \mathsf{dk})$ generated by $\mathcal{K}$, and for all messages m, $\mathcal{D}(\mathsf{dk}, \mathcal{E}(\mathsf{ek}, m)) = m$. Classical security notions for encryption are $\mathsf{IND}-\mathsf{CPA}$ and $\mathsf{IND}-\mathsf{CCA}$, where the adversary tries to distinguish the ciphertext of two messages of its choice, being given just the encryption key, or also access to the decryption oracle, respectively.

Commitment Schemes. In a commitment scheme, a *sender* commits on a message m to a *receiver* without revealing any information, but with the guarantee that at the *opening* time, a unique message can be revealed. There are two basic properties: the commitment must be *hiding*, which guarantees that no information about m is leaked during the *commit* phase, and be *binding*, which guarantees that only one message can be revealed during the *opening* phase. Additional classical properties are extractability, equivocability, and non-malleability.

4.2.4 Concrete Encryption Schemes

ElGamal Encryption. Introduced in 1985, by ElGamal [ElG85], based on the CDH assumption, and achieving $\mathsf{IND}-\mathsf{CPA}$ security under the DDH, the ElGamal encryption scheme works as follows:

$\mathsf{Key\ Generation}$: Let $x \in \mathbb{Z}_q$ the decryption key, the associated encryption key is $y = g^x$;

$\mathsf{Encryption}$: Given a message $m \in \mathbb{G}$, let choose $r \xleftarrow{\$} \mathbb{Z}_q$, then compute $u = g^r$ and $v = y^r m$. The ciphertext is $c = (u, v)$;

$\mathsf{Decryption}$: Given a ciphertext $c = (u, v)$, the message can be decrypted as $m = v \cdot u^{-x}$.

More precisely, within time t:

$$\mathsf{Adv}^{\mathsf{ind-cpa}}(t) \leq 2 \times \mathsf{Adv}^{\mathsf{ddh}}(t).$$

Cramer-Shoup Encryption. The Cramer-Shoup encryption scheme [CS98] achieves $\mathsf{IND}-\mathsf{CCA}$ security under the DDH assumption:

$\mathsf{Key\ Generation}$: Let $g_1, g_2 \xleftarrow{\$} \mathbb{G}$ and $x_1, x_2, y_1, y_2, z \xleftarrow{\$} \mathbb{Z}_q$. Let $c = g_1^{x_1} g_2^{x_2}$, $d = g_1^{y_1} g_2^{y_2}$, $h = g_1^z$ and a hash function H, chosen from the family of universal one-way functions. The public key is (g_1, g_2, c, d, h, H) and the private key is (x_1, x_2, y_1, y_2, z);

Encryption: Given a message $m \in \mathbb{G}$, let choose $r \xleftarrow{\$} \mathbb{Z}_q$, then compute $u_1 = g_1^r$, $u_2 = g_2^r$, $e = h^r m$, $\alpha = H(u_1, u_2, e)$, and $v = c^r d^{r\alpha}$, the ciphertext is $c = (u_1, u_2, e, v)$;

Decryption: Given a ciphertext $c = (u_1, u_2, e, v)$, one first computes $\alpha = H(u_1, u_2, e)$ and checks whether $u_1^{x_1+y_1\alpha} u_2^{x_2+y_2\alpha} = v$ or not. If this condition does not hold, then it rejects, otherwise it outputs $m = e/u_1^z$.

Such an $\mathsf{IND} - \mathsf{CCA}$ encryption scheme can be used as a perfectly binding commitment scheme. The decryption key allows extractability and the $\mathsf{IND} - \mathsf{CCA}$ security level makes the commitment scheme non-malleable, but also extractable while still (computationally) hiding.

More precisely, within time t and after at most q_d decryption queries:

$$\mathsf{Adv}^{\mathsf{ind-cca}}(t) \leq 2 \times \mathsf{Adv}^{\mathsf{ddh}}(t) + \mathsf{Succ}_H^{\mathsf{2nd}}(t) + 3q_d/q.$$

4.3 Security Model

In order to analyze the security of PPSS protocols, we first provide a formal description of the security model. This is a game-based security definition, in the same vein as [BR94, BR95] for key distribution schemes and [BPR00] for password-authenticated key exchange. It adapts the PPSS definition from [BJSL11] and the security model from [JKK14]. We define security in terms of a *key derivation mechanism* or indistinguishability of the actual secret from a random one, as in [JKK14], since our goal is to later use the secret as a symmetric key. In particular, we do not want to rely on a PKI or any authenticated public values, hence our model description is similar to security models for PAKE.

4.3.1 Password-Protected Secret Sharing

We first describe the participants and the two steps of a PPSS protocol.

Participants and Parameters. We assume a fixed set of participants involved in the protocol, each of which is either a user or a server. The set of all participants is the union of the nonempty disjoint and finite sets, $\mathtt{User} \cup \mathtt{Server}$.

Each user $U \in \mathtt{User}$ holds two threshold values t_ℓ and t_r, where t_r is the number of shares required to *recover* the secret and t_ℓ the maximum number of shares that can be known without *leaking* any information about the secret, as well as some password pw chosen independently and uniformly from a dictionary $\mathcal{D}$ of cardinality $\#\mathcal{D}$.

Each server $S \in \mathtt{Server}$ holds a secret key sk, and possibly an associated public key pk. However we stress that even if there is a public key pk, authenticity cannot be assumed *a priori* during the reconstruction phase since users will just have to remember their passwords and nothing else that would be required to authenticate additional data.

Initialization. The goal of the user U is to generate a key K so that he later can recover it with the help of t_r servers among n available servers, just using his password. He thus runs an initialization protocol with n servers, using their public keys, his password and some random coins. He ends up with a random key K and some additional information PInfo: nobody else than U has any information about K, however PInfo can be made public.

Secret Reconstruction. While the initialization phase assumes that all the servers are honest, the public keys are authentic, and the data are not modified during the communication, for the reconstruction phase, the adversary controls the network and can forward, alter, delay, replay, or delete any message. The adversary can also provide fake public data: nothing is authenticated anymore!

Anyway, just using his password, the user U should be able to recover K, with the help of the servers, in a verifiable/robust way, even if some public keys in PInfo are not guaranteed to be correct.

Each participant (either user or server) can run several executions of the protocol, possibly concurrently, we thus denote an instance i of player P as P^i. Each instance may be activated once only: the adversary is given oracle accesses to interact with all the user's and server's instances that are stateful interactive polynomial-time Turing machines.

4.3.2 The adversarial model

During the reconstruction phase, the adversary is given total control of the network: it can forward, alter, delay, replay, or delete any message sent by any player. To model this ability, it is given access to the following oracles:

- $\mathsf{Execute}(U^i, \{S_k^{j_k}\})$: This query models a passive attack. This makes an instance U^i to interact with several instances of servers $\{S_k^{j_k}\}$ as they would do during the reconstruction protocol. The adversary eventually gets back the entire transcript;

- $\mathsf{Send}(P^i, m)$: This query models an active attack. This sends a message m to the instance P^i. This message m can be a fresh message, or a replay, a forward, etc. A specific message Start_k^j to a user's instance U^i makes it initiate a communication with the server's instance S_k^j.

The security goal is to guarantee the privacy of the secret key K reconstructed by the user. This is usually modeled by an indistinguishability game, with access to a Test-query, where b is a global secret random bit:

- $\mathsf{Test}(U^i)$: This query characterizes the indistinguishability of the key K computed by instance U^i. If this instance has not yet completed the reconstruction, the answer is UNDEFINED ; if the reconstruction failed, the answer is $\perp$; otherwise, the answer is either the real reconstructed value if $b = 1$ or a random one (always the same for user U, but independent of the real one) if $b = 0$.

The adversary eventually outputs its guess b' for the bit b, to show its ability to distinguish real multiple executions of the protocol from ideal executions: one can note that in the random case ($b = 0$), which models the ideal executions, a user U always terminates with the same key, or fails. This means that the adversary should not be able to make him accept a different key.

In addition to control the network and the communications, the adversary can corrupt servers, and get back their secret keys, due to, e.g., a poorly-administered server, compromise of a host computer, or cryptanalysis. This is modeled by the $\mathsf{Corrupt}$-query:

- $\mathsf{Corrupt}(S_k)$: This outputs the secret key sk_k of the server S_k.

4.3.3 Semantic Security

Definition. Once the initialization phase is completed for many users, with random passwords uniformly and independently drawn from a dictionary $\mathcal{D}$, the security game models the indistinguishably of the secret keys, a.k.a. *semantic security*, the adversary can ask as many oracle queries (Execute, Send, Test, and Corrupt), as it wants, in any order it wants, in order to guess the bit b: it outputs its guess b'. We measure the quality of an adversary $\mathcal{A}$ by its advantage

$$\mathsf{Adv}(\mathcal{A}) = \Pr[b' = 1|b = 1] - \Pr[b' = 1|b = 0] = 2 \times \Pr[b' = b] - 1.$$

Trivial Attacks. Two kinds of "on-line dictionary attacks" are unavoidable:

- if the adversary guesses the correct password, it will be able to reconstruct the actual secret K after q_c corruption queries and $t_r - q_c$ interactions with honest servers. Even after just $t_\ell - q_c$ interactions, it may come up with t_ℓ shares, which may leak some information about the actual secret key: it thereafter asks for an Execute-query, and tests the instance involved in this session, to distinguish the real case from the random case. Its success probability is however upper-bounded by $q_s/(t_\ell - q_c) \times 1/\#\mathcal{D}$, where q_s is the number of server instances involved during the attack, q_c the number of Corrupt-queries, and $\#\mathcal{D}$ the size of the password dictionary.

- whereas the initialization phase was assumed to be done with authentic server public keys, for the reconstruction phase, the adversary can send totally fake public keys in PInfo that it generated itself from a randomly chosen password pw. It thus also knows the secret keys and can simulate the view of the user by emulating all the servers. If the password guess was correct, the user should successfully terminate, whereas a wrong guess would lead to inconsistent information. Its success probability is therefore upper-bounded by $q_u/\#\mathcal{D}$, where q_u is the number of user instances involved in the attack.

4.3.4 Secure PPSS

As a consequence, we will say a (t_r, n)-PPSS scheme is (t_ℓ, ε, t)-secure if for any adversary $\mathcal{A}$, running within time t, asking at most $q_c < t_\ell$ Corrupt-queries and invoking at most q_u user instances and q_s server instances,

$$\mathsf{Adv}(\mathcal{A}) \leq \frac{1}{\#\mathcal{D}} \times \left(\frac{q_s}{t_\ell - q_c} + q_u \right) + \varepsilon.$$

In [JKK14], they proposed such a protocol that achieves the optimal t_ℓ-security, for $t_\ell + 1 = t_r$, but at the cost of verifiable oblivious pseudorandom functions. Our goal is to build much more efficient protocols, possibly lowering the security level: $t_\ell = 3t_r - 2n - 1$.

Before going into more details about our constructions, let us review the required or expected properties for a PPSS, initiated with a password pw and secret K for a user U. Some are already covered by the security model, some offer additional features:

- **Correctness.** To be viable, a password-protected secret sharing must guarantee that at least t_r honest servers should allow the user that plays with his password pw to recover his secret K.

- **Soundness.** As already guaranteed by our security model, when a user terminates with a key K', this is the correct key K. More precisely, when playing with the correct password pw, the user ends up with $K' \in \{K, \perp\}$ without any assumption about the communications and the server behaviors: there are no authenticated channels nor any authenticated data.

- **Robustness.** Due to our communication model, messages can be lost, modified, or even totally faked by the adversary. Of course, one cannot avoid Denial-of-Service (DoS) attacks, since the adversary can simply block any communication. However, an important property, already required by [JKK14], is the so-called *robustness*: even if the adversary alters many messages, as soon as t_r communications with servers are unmodified the user can *efficiently* recover its secret.

The general issue with robustness is that when the user has interacted with n servers but only t_r shares are valid, the cost of trying all the t_r subsets is exponential! In [JKK14], they addressed this issue by making some inner protocols secure against malicious servers, with additional zero-knowledge proofs of honest behavior, but this is at a high communication cost. Our goal is to provide this property at a much lower cost.

4.4 A Robust Gap Threshold Secret Sharing Scheme

Our technique is generic, and so we start from any threshold secret sharing scheme, with two algorithms `ShareGen` and `Reconstruct` that respectively share a secret and reconstruct it. One can for example use the classical Shamir's secret sharing scheme [Sha79] to which we will add this new robustness feature, at the cost of having a threshold gap secret sharing scheme that is enough to get a robust PPSS scheme (for details about secret sharing schemes see 4.2.3).

Let us first give the intuition of the technique, and we then explain how we can realize it in practice.

4.4.1 Intuition

The valid shares are denoted $(s_1, \ldots, s_n)$ and the fingerprints of these shares $(\sigma_1, \ldots, \sigma_n)$. At the same time of the share distribution, the product $\mathcal{S}$ of all fingerprints modulo an integer N is published. In order to reconstruct the secret, having received m candidate shares, one computes its fingerprints $(\tau_1, \ldots, \tau_m)$ and the product of them $\mathcal{T} = \prod \tau_i$. The ratio $\mathcal{T}/\mathcal{S} \bmod N$ will cancel out the fingerprints of all the correct share values leading to the ratio $\mathcal{T}'/\mathcal{S}' \bmod N$, where $\mathcal{S}'$ is the product of the fingerprints of the valid shares that the receiver does not have in the list of candidates and $\mathcal{T}'$ the product of the fingerprints of the candidates that are invalid. From $\mathcal{S}'$, one could easily check for every candidate, whether it is in this product or not, and therefore identify which candidate is correct or not.

Of course, $\mathcal{S}'$ has to be computed with good precision to allow the last verification, but not too much in order to avoid individual checks or any unnecessary leakage of information. The computations are thus performed modulo N, for a well-chosen value.

4.4.2 Description

We now explain how one can detect the valid shares when the fingerprints are either correct or random.

Initialization. We assume we have a set of n initial values $(s_1, \ldots, s_n)$, and their k-bit string fingerprints $(\sigma_1, \ldots, \sigma_n)$. As fingerprint function we use a hash function $F : \{0,1\}^* \to \{0,1\}^k$ modeled as a random oracle.

In the following, we will be given a set of m candidate shares, whose fingerprints are $(\tau_1, \ldots, \tau_m)$: these fingerprints are either correct (the same as in the list $(\sigma_1, \ldots, \sigma_n)$ or random for incorrect candidate shares). From this set of candidate shares, if at least t_r are correct, we want to efficiently identify the correct values (to *recover* the secret in a threshold secret sharing scheme, hence the r-subscript in t_r). However, if at most t_ℓ are correct, the protocol should not *leak* any information about which candidates are valid and which are not (hence the ℓ-subscript in t_ℓ).

From the initial set $(\sigma_1, \ldots, \sigma_n)$ of size n and the threshold t_r, one chooses a prime number N such that $2^{2k(n-t_r)+1} < N \leq 2^{2k(n-t_r)+2}$, computes the product $\mathcal{S} = \prod_{i=1}^{n} \sigma_i \bmod N$, and publishes $\mathsf{SSInfo} = (\mathcal{S}, N)$.

Reconstruction. Given the $\mathsf{SSInfo} = (\mathcal{S}, N)$ and fingerprints $(\tau_1, \ldots, \tau_m)$ of the $m \leq n$ candidates, which are either correct (at least t_r of them) or random (all the other ones), one computes the ratio $\gamma = \prod_{i=1}^{m} \tau_i / \mathcal{S} \bmod N$, which can be written as $\gamma = \mathcal{T}'/\mathcal{S}' \bmod N$, where $\mathcal{T}'$ is the product of the fingerprints of the invalid candidates and $\mathcal{S}'$ the product of the fingerprints of the values that are not in the list of the candidates, both over the integers. Then, we know that $\mathcal{T}' < 2^{k(m-t_r)} \leq 2^{k(n-t_r)}$ and $\mathcal{S}' < 2^{k(n-t_r)}$.

Our experimental results (for details, see 4.4.3) show that on one hand, approximately one half of the cases $\mathcal{T}'$ and $\mathcal{S}'$ are coprime. On the other, both values share some small factors. We denote $\mathcal{T}''/\mathcal{S}''$ as the irreducible fraction where all the small common factors of $\mathcal{T}'/\mathcal{S}'$ were canceled out.

Using the following result from [FSW03], we can recover the $\mathcal{T}''/\mathcal{S}''$ of $\gamma = \mathcal{T}'/\mathcal{S}' = \mathcal{T}/\mathcal{S}$, with $\mathcal{T}'' \leq \mathcal{T}' < 2^{k(n-t_r)}$ and $\mathcal{S}'' \leq \mathcal{S}' < 2^{k(n-t_r)}$, under appropriate conditions.

Theorem 4.1. (Numerical Rational Number Reconstruction) *Let $z = \frac{x}{y} \bmod N$ such that $-X \leq x \leq X$ and $0 < y \leq Y$. If N is relatively prime to y and $2XY < N$ then the solution is unique and it is possible to recover x and y efficiently by using two-dimensional lattice theory.*

Considering $X = 2^{k(n-t_r)} - 1$ and $Y = 2^{k(n-t_r)} - 1$, we indeed have $2XY \leq 2(2^{k(n-t_r)} - 1)(2^{k(n-t_r)} - 1) < N$ and $X > 0$, $Y > 0$, hence we can efficiently recover $\mathcal{T}''$ and $\mathcal{S}''$ from γ.

Now, if τ_i is the fingerprint of a valid share, it should be canceled out in $\mathcal{T}'$, but there might still be some small factors in common between τ_i and $\mathcal{T}''$. On the other hand, if τ_i is the fingerprint of a random invalid share, it should not be completely canceled out in $\mathcal{T}'$. However, there is still a chance that some small factors have been canceled out, leading to $\mathcal{T}''$ in the irreducible form.

Hence, our decision algorithm is the following one: we denote t_i the bit size of $|\gcd(\mathcal{T}'', \tau_i)|$; if $t_i \geq k/2$, this is an invalid share, otherwise this is a valid share. In Figure 4.1, we present experimental results that validate this decision algorithm for $k = 128$-bit. It is possible to see clearly that for a valid τ_i, t_i is a small number (half of them equal to 1) and for an invalid τ_i, t_i is a big number (44% of them is equal to k). We

have computed 2^{21} times the value of $\gcd(\mathcal{T}'', \tau_i)$ and in case of Figure 4.1a, the highest bit size of t_i is 35 (much less than 64). On the other hand, in Figure 4.1b the least value is 96 (much more than 64). Both with probability 1 over 2^{21}. A more fine analysis with different sizes of the fingerprint and the number of shares can be found in 4.4.3.

Information Leakage. On the opposite, we would like to evaluate the information leaked by $\mathcal{S}$ when there are at most t_ℓ valid values. More precisely, given $\mathcal{S}$, is it possible to distinguish t_ℓ valid values from t_ℓ random values?

For a t_r-threshold secret sharing scheme, the entropy of the tuple $(\sigma_1, \ldots, \sigma_n)$ is $k(t_r - 1)$. Since $\mathcal{S}$ reveals the product modulo N, with $N < 2^{2k(n-t_r)+2}$, the remaining entropy on the shares is at least $k(t_r - 1) - 2k(n - t_r) - 2 = k(3t_r - 2n - 1) - 2$. If this is greater than kt_ℓ, no one can distinguish t_ℓ random values from t_ℓ correct values for the shares: we thus need $k(3t_r - 2n - 1) - 2 \geq kt_\ell$. When $k > 2$, this essentially means $t_\ell \leq 3t_r - 2n - 1$: by choosing $t_\ell = 3t_r - 2n - 1$, we are safe. For example, one can take $t_r = \lceil 3n/4 \rceil$ and $t_\ell = \lfloor n/4 \rfloor$.

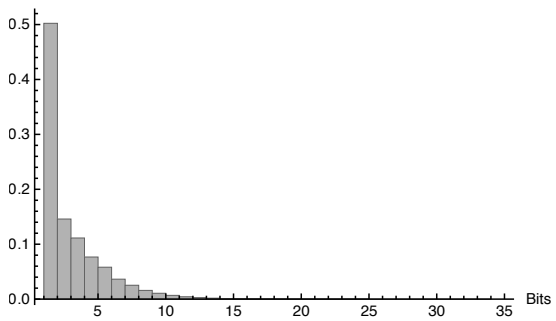
4.4.3 Experimental Results

We have implemented the decision algorithm to validate the idea of taking the $|\gcd(\mathcal{T}'', \tau_i)| \geq k/2$. We have tested 2^{21} random shares for $k = \{64, 96, 128, 256\}$-bit fingerprints and for $n = \{32, 44, 60, 92\}$ shares to evaluate the distribution of large prime numbers in different settings. In Table 4.1 we present the probabilities for the best cases, which are: (i) $\gcd(\mathcal{T}'', \tau_i) = 1$ (coprime) when τ_i is correct, meaning that all common factors are canceled out and (ii) $|\gcd(\mathcal{T}'', \tau_i)| = k$ (no factors were canceled out) when τ_i is incorrect. We can remark these probabilities are between 40% and 50%.

Table 4.1: Probabilities of the best cases (i.e., probability that $\gcd(\mathcal{T}'', \tau_i) = 1$ when τ_i is correct and probability that $|\gcd(\mathcal{T}'', \tau_i)| = k$ when τ_i is random)

n	k							
	64		96		128		256	
	τ_i corr.	τ_i rand.	τ_i corr.	τ_i rand.	τ_i corr.	τ_i rand.	τ_i corr.	τ_i rand.
32	49, 24%	45, 12%	49, 62%	49, 68%	49, 68%	44, 70%	50, 05%	44, 55%
44	75, 75%	43, 78%	47, 93%	43, 62%	47, 66%	43, 20%	47, 10%	43, 88%
60	45, 51%	42, 41%	45, 68%	42, 24%	42, 75%	45, 92%	45, 92%	41, 95%
92	43, 05%	40, 93%	42, 76%	41, 15%	43, 08%	40, 77%	43, 34%	40, 58%

(a) $\gcd(\mathcal{T}'', \tau_i)$-bitlength for valid τ_i.



(b) $\gcd(\mathcal{T}'', \tau_i)$-bitlength for invalid τ_i.

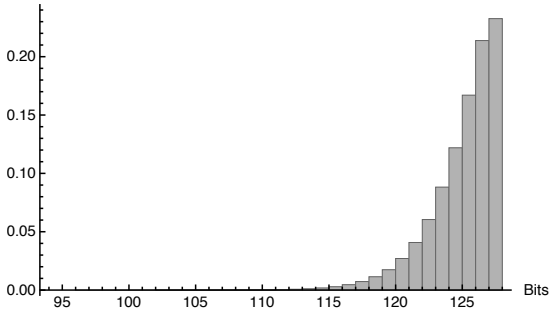


Figure 4.1: Length in bits of $\gcd(\mathcal{T}'', \tau_i)$ for a fingerprint of size 128-bits and 32 shares

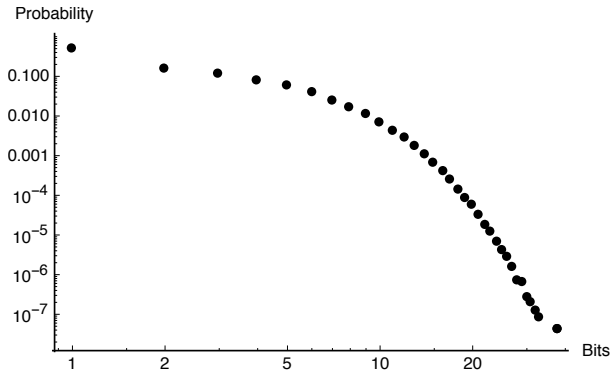
On the other hand, in Table 4.2 we present our worst-cases of the decision algorithm. One can see that for $k = 64$ the algorithm fails, with both too large $\gcd(\mathcal{T}'', \tau_i)$ when τ_i is correct and too small $\gcd(\mathcal{T}'', \tau_i)$ when τ_i is random, leading to false positive and false negative decisions. This is due to the too small fingerprints. Indeed, increasing k to 96, the probability of false positive/negative decisions is drastically reduced: the worst cases are far from $k/2$, even for $n = 92$. No bad decisions are taken among the millions of tests.

Table 4.2: Bit size t_i of $\gcd(\mathcal{T}'', \tau_i)$ for different sizes k of the fingerprint and numbers n of shares

n	k							
	64		96		128		256	
	τ_i corr.	τ_i rand.	τ_i corr.	τ_i rand.	τ_i corr.	τ_i rand.	τ_i corr.	τ_i ran
32	**33**	**29**	35	65	35	96	34	223
44	**41**	**26**	35	60	40	92	36	221
60	**39**	**27**	38	61	40	94	39	220
92	**44**	**25**	43	53	53	84	43	217

In the Figure 4.2 we present the distribution of $t_i = |\gcd(\mathcal{T}'', \tau_i)|$ for $k = 96$ and $n = 32$, when τ_i a valid fingerprint. It is possible to see that there is a high probability of t_i be equals to 1 ($\mathcal{T}''$ and τ_i coprimes) and the probability is reducing while the bit size is increasing. Our experiments suggest that, for our setting, the probability of bit-lengths is bounded by $2^{-x/2}$. In our case ($k = 96$ and $n = 32$), the probability of deciding a τ_i as valid while it is not (false positive) looks approximately 2^{-28}.

Figure 4.2: Distribution of the bit-length of $\gcd(\mathcal{T}'', \tau_i)$ for correct τ_i, when $k = 96$ and $n = 32$



4.5 Password-Protected Secret Sharing Protocol

Thanks to our new (t_ℓ, t_r, n)-RGTSSS, we do not need to use a VOPRF. With a classical threshold secret sharing scheme, as in [JKK14], this is not possible to avoid the verifiability of the OPRF. This verifiability is at the cost of zero-knowledge proofs of honest behavior of the servers. We can now describe our general structure of PPSS protocol, using an OPRF as black-box.

We thereafter provide two instantiations, with two appropriate OPRFs, in the same vein as the ones proposed in [JKK14], using similar computational assumptions (see 4.2.3):

- the first OPRF relies on the CDH evaluation, similar to the protocol 2HashDH from [JKK14], but without NIZKs. It leads to a PPSS construction quite similar to [JKKX16].
- the second OPRF is an oblivious evaluation of the Naor-Reingold PRF [NR97]. Then, in the PPSS, the gain of the zero-knowledge proofs by the server is quite significant, but we still need some proofs from the user, to ensure the input is in the correct domain, otherwise there is no guarantee on the PRF property.

4.5.1 General Description

As already presented in the high-level description, our protocols are in two phases: the initialization phase which is assumed to be executed in a safe environment with reliable communications and correct inputs from the servers, and the reconstruction phase during which the password only is considered correct, while all the other inputs can be faked by the adversary.

4.5.1.1 Initialization.

We assume that each server S_k owns a key pair $(\mathsf{sk}_k, \mathsf{pk}_k)$ that defines a PRF F_k, with public parameters defined by pk_k and a secret key defined by sk_k, that admits an OPRF protocol to allow a user with input m to evaluate $F_k(m)$ without leaking any information on m to the server. We additionally use a (t_ℓ, t_r, n)-robust gap threshold secret sharing scheme, where we can assume that $t_r = 3n/4$ and $t_\ell = n/4 - 1$ (which is compatible with our previous construction), and a commitment scheme Commit (see 4.2.3). The user U first chooses a secret password pw:

1. the user interacts with n servers to obliviously evaluate $\pi_k = F_k(\mathsf{pw})$, and $\Pi = (\mathsf{pk}_k)_k$ is the tuple of the public keys of the involved servers;
2. for a random value $R = K\|r$, where K is the random secret key the user wants to share and r some random coins. The user generates $(s_1, \ldots, s_n, \mathsf{SSInfo}) \leftarrow \mathtt{ShareGen}(R)$, so that any subset of t_r shares among $\{s_1, \ldots, s_n\}$ can efficiently reconstruct R;
3. then, the user builds $\sigma_k = \pi_k \oplus s_k$, for $k = 1, \ldots, n$, and sets $\Sigma = (\sigma_k)_k$;
4. the user generates $C = \mathsf{Commit}(\mathsf{pw}, \Pi, \Sigma, \mathsf{SSInfo}, K; r)$. We denote by $\mathsf{PInfo} = (\Pi, \Sigma, \mathsf{SSInfo}, C)$ the public information that the user will need later to recover its secret K;
5. the user thus gives PInfo to all the servers.

We stress that during this initialization phase, all the values of Π are the real public keys and $(\pi_k)_k$ are the correct evaluations of the PRFs. On the opposite, during the reconstruction phase, all the values in PInfo will be provided by the servers, but through the adversary, who might alter them.

4.5.1.2 Reconstruction.

For the reconstruction, the user interacts with at least t_r servers, that provide him $\mathsf{PInfo} = (\Pi, \Sigma, \mathsf{SSInfo}, C)$, and help him to compute $\pi_k = F_k(\mathsf{pw})$ for several values of k, using pk_k from Π. No information is trusted anymore, and so the reconstruction phase perform several verifications:

1. the user first limits the oblivious evaluations of $\pi_k = F_k(\mathsf{pw})$ to the servers that sent the same majority tuple $\mathsf{PInfo} = (\Pi, \Sigma, \mathsf{SSInfo}, C)$. If the number of such servers is less than t_r, one aborts with $K \leftarrow \perp$;

2. for all these π_k (or similarly, all the k he kept), the user computes $s_k = \sigma_k \oplus \pi_k$, using σ_k from Σ (from PInfo);

3. using these $\{s_k\}$ with at least t_r correct shares, and SSInfo (from PInfo), with RGTSSS, the user reconstructs the shared secret R (or aborts with $K \leftarrow \perp$ if the reconstruction fails);

4. the user parses the secret R as $K \| r$, and checks, from PInfo, whether $C = \mathsf{Commit}(\mathsf{pw}, \Pi, \Sigma, \mathsf{SSInfo}, K; r)$;

5. if the verification succeeds, K is the expected secret key, otherwise the user aborts with $K \leftarrow \perp$.

4.5.2 Protocol I: One-More-Gap-Diffie-Hellman-based PRF

Our first instantiation is based on CDH-like assumptions in the random oracle model. The arithmetic is in a finite cyclic group $\mathbb{G} = \langle g \rangle$ of prime order q. We need a full-domain hash function H_1 onto $\mathbb{G}$, and another hash function H_2 onto $\{0,1\}^{\ell_2}$. Since we already are in the random oracle model for the PRF, we can implement the commitment scheme with a simple hash function H_3 onto $\{0,1\}^{\ell_3}$: $C = \mathsf{Commit}(\mathsf{pw}, \Pi, \Sigma, \mathsf{SSInfo}, K; r) := H_3(\mathsf{pw}, \Pi, \Sigma, \mathsf{SSInfo}, K, r)$, which allows a better efficiency.

For a private key $\mathsf{sk} = x \in \mathbb{Z}_q$, we consider the pseudorandom function $F_x(m) = H_2(m, g^x, H_1(m)^x)$, for any bitstring $m \in \{0,1\}^*$, where the public key is $\mathsf{pk} = y = g^x$. In 4.5.2.1, we prove this is indeed a PRF. In addition, it admits an oblivious evaluation, that does not leak any information, thanks to the three simulators $\mathcal{S}im$, $\mathcal{S}im_U$ and $\mathcal{S}im_S$, as presented in Figure 4.3: $\mathcal{S}im$ simulates an honest transcript, $\mathcal{S}im_U$ simulates an honest user interacting with a malicious server, and $\mathcal{S}im_S$ simulates an honest server with a malicious user.

These simulators will be used by our simulator in the full security proof. They generate perfectly indistinguishable views to the adversary, but they require $\mathsf{CDH}_g(y, \cdot)$ and $\mathsf{DDH}_g(y, \cdot, \cdot)$ evaluation, and thus oracle access when the secret keys are not known. Since the indistinguishability of the PRF relies on the $\mathsf{CDH}_g(y, \cdot)$ assumption, the overall

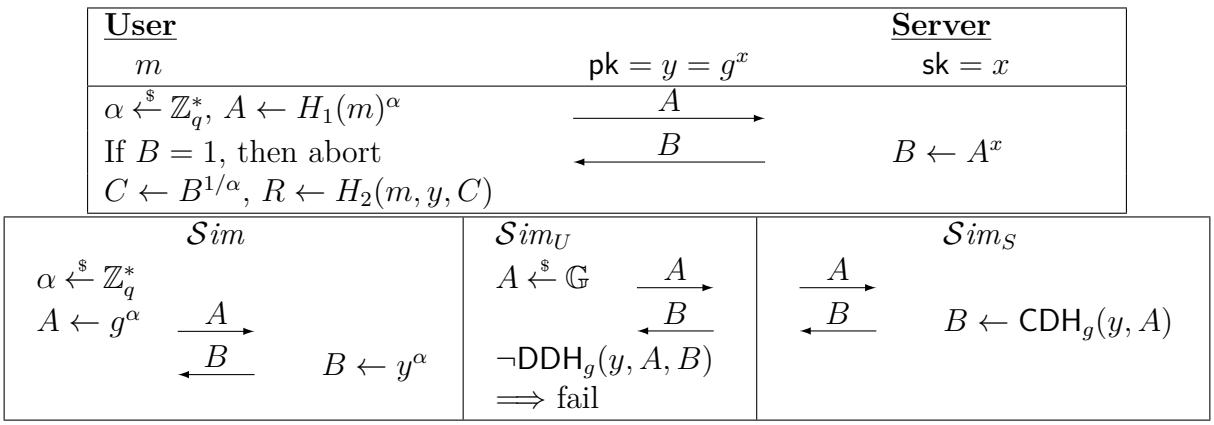


Figure 4.3: Secure Oblivious Evaluation of the PRF based on OMGDH

security relies on the One-More Gap Diffie-Hellman (OMGDH) assumption (see 4.2.2) as shown in the last step of the proof.

Theorem 4.2. *For any adversary $\mathcal{A}$, against the Protocol I, that corrupts no more than q_c servers, involves at most q_s instances of the servers, q_u instances of the user, and asks at most q_1, q_2, q_3 queries to H_1, H_2, H_3, respectively*

$$\mathsf{Adv}(\mathcal{A}) \leq \left(q_u + \frac{4q_s}{n - 4q_c}\right) \times \frac{1}{\#\mathcal{D}} + n \times \mathsf{Succ}^{\mathsf{omgdh}}(q_1, q_s, t, n \cdot q_u + q_2) + (q_3^2 + 2) \cdot 2^{-\ell_3}.$$

4.5.2.1 Security Proof.

The complete and detailed proof of the Theorem is given in 4.5.2.1. The rough idea is the following: in the real attack game, we focus on a unique user, against a static adversary (the corrupted servers are known right after the initialization, and before any reconstruction attempt). All the parameters are honestly generated, the simulator knows the secret informations to answers the queries, and two random keys K_0 (random) and K_1 (real), as well as a bit b, are selected randomly to answer Test-queries. In the final game, we simulate all the answers to the adversary without using a password. A random value will be chosen at the very end of the simulation and used as a password in order to decide if some bad events should have occurred, which will immediately upper-bound the advantage of the adversary.

We first modify the way Execute-queries are answered, using $\mathcal{S}im$ that perfectly simulates honest transcripts user-servers, and we set user's key to K_1.

Then, we deal with Send-queries to the honest user, trying to exclude the cases of a fake public information PInfo' (sent by the majority of servers): first, we do as before if the commitment C' in PInfo' is different from the expected value C generated during the initialization, but eventually we set $K \leftarrow \perp$. This would just make a difference for the adversary if C' indeed contains the good password pw, which is defined as the event PWinC. This event PWinC can be evaluated using the list of queries asked to H_3. Then, a similar argument applies when a wrong PInfo' is sent, but with a correct C, under the binding propriety of the commitment H_3.

Once we have fixed this, and we trust the public values, we can use $\mathcal{S}im_U$, that perfectly simulates a flow A from the user to a server, and can decide on the honest behavior of the servers. Then $\mathcal{S}im_U$ accepts with $K \leftarrow K_1$ in the honest case or aborts with $K \leftarrow \perp$ otherwise. Hence, we remark that we answer Send-queries without calling the H_1 or H_2 oracles, but just using K_1, and no secret sharing reconstruction is used anymore.

Next step is to replace all the shares in the initialization phase by random and independent values. We know that until the adversary does not get more than $t_\ell = n/4$ of these shares, it cannot detect whether they are random or correct. We define the event PWinF to be the bad event, where the adversary has enough evaluations of the PRF to notice the change. Again, our simulator is able to decide the event PWinF by checking whether pw has been queried with the right inputs to H_2, and how many times. We eventually replace the hash value C in the initialization phase by a random C.

One can note that, in the end, the password pw is not used anymore during the simulation, but just to determine whether the events PWinC or PWinF happened. In addition, K_1 does not appear anymore during the initialization phase, hence one cannot make any difference between K_0 and K_1: $\mathsf{Succ}_{\mathcal{A}} = 1/2$ in the last game. As a consequence, $\mathsf{Adv}(\mathcal{A}) \leq \Pr[\mathsf{PWinC}] + \Pr[\mathsf{PWinF}] + \varepsilon$, where ε comes from the collisions or guess on the

random oracles. To evaluate the two events PWinC or PWinF to happen, we choose a random password pw at the very end only: Pr[PWinC] is clearly upper-bounded by $q_u/\#\mathcal{D}$, since q_u is the maximal number of fake commitment attempts containing the right pw that can be different from the expected ones; PWinF means that the adversary managed to get $n/4 - q_c$ evaluations of the PRFs under the chosen pw, since it can evaluate on its own the values under the q_c corrupted servers. But unless the adversary gets more evaluations than the number q_s of queries asked to the servers (which can be proven under the OMGDH assumption), the number of *bad* passwords (for which the knows at least $n/4 - q_c$ evaluations of the PRFs) is less than $q_s/(n/4 - q_c)$. So the probability that the chosen pw is such a bad password is less than $q_s/(n/4 - q_c) \times 1/\#\mathcal{D}$.

4.5.3 F_x is a PRF

Lemma 4.3. *The above function F_x is a PRF under the Computational Diffie-Hellman (*CDH*) assumption.*

Given an instance $(g, y = g^x, h)$, one wants to compute $h^x = \mathsf{CDH}_g(y, h)$. Any H_1-query on a new m is answered by h^z, for a random scalar z, and the tuple (m, z) is stored in the list Λ_1. For any PRF evaluation on a new m, one first asks for $H_1(m)$, chooses a random value $r \in \{0,1\}^\ell$, answers r and stores (m, z, r) in the list Λ_F. For any new H_2-query (m, y, H), one first asks for $H_1(m)$, and answers by a random value. A difference happens here from the real case if $H = \mathsf{CDH}_g(y, H_1(m))$ and (m, z, r) is in Λ_F, since the answer should be r, and not a random value. The same problem happens if the F query is asked later. In both cases, at the end of the game, among all the H values from the H_2-queries and the $(m, z, r) \in \Lambda_F$, one pair (H, z) satisfies $H = \mathsf{CDH}_g(y, H_1(m)) = \mathsf{CDH}_g(y, h^z) = \mathsf{CDH}_g(y, h)^z$. By choosing it at random, one gets $\mathsf{CDH}_g(y, h)$ with non-negligible probability.

4.5.4 Security Proof of the Protocol I

For the proof we consider an adversary as the one defined in the security model description in Section 4.3. After the initialization phase, this adversary can ask as many Execute and Send-queries, Test-queries and also Corrupt-queries as it wishes, and has access to the extra random oracles H_1, H_2, and H_3.

The proof will be performed by a sequence of games, starting from the real indistinguishability game, focusing on a unique user, against a static adversary (the corrupted servers are known right after the initialization, and before any reconstruction attempt). In the final game, the goal to achieve is to simulate all the queries to the adversary without using a password. A random value will be chosen at the very end of the simulation and used as a password in order to decide if some bad event should have occurred, which will immediately upper-bound the advantage of the adversary.

Game $\mathbf{G_0}$: This initial game ponds to the real attack game, in the random oracle model. Three oracles are available to the adversary, H_1, H_2, and H_3 and the adversary chooses some servers to be corrupted: the related secret informations are then revealed to the adversary right after the initialization.

First, we emulate the initialization phase, which is honestly performed: we choose one random pw, n random keys $(x_k)_k$ for the servers' secret information, which lead to the evaluation of $(\pi_k)_k$, together with their public part $\Pi = (y_k = g^{x_k})_k$, and

one random value corresponding to the secret K, together with a secret sharing $(s_1, \ldots, s_n, \mathsf{SSInfo})$ of $R = K \| r$, for a random r. This last random value r is used to compute the commitment $C = H_3(\mathsf{pw}, \Pi, \Sigma, \mathsf{SSInfo}, K, r)$, where $\Sigma = (\sigma_k = \pi_k \oplus s_k)_k$. One also chooses a second random key K_0, as well as a bit $b \xleftarrow{\$} \{0,1\}$, both used in Test-queries: in a reconstruction execution, if a key K_1 is reconstructed, the Test-query outputs K_b, if the reconstruction is not completed or failed, the Test-query outputs UNDEFINED or $\perp$. For the reconstruction, we simulate all the instances, the user and the servers, in Execute and Send-queries, as the real players would do.

The adversary eventually outputs its guess b' for the bit b. The output of the game is the success bit $S = (b' = b)$. By definition we have :

$$\mathsf{Succ}_{\mathbf{G}_0} = \Pr[S] \qquad\qquad \mathsf{Adv}(\mathcal{A}) = 2 \times \mathsf{Succ}_{\mathbf{G}_0} - 1$$

Game $\mathbf{G}_1$: We do not modify the initialization, and first deal with Execute-queries, by replacing the user and the servers by the simulator $\mathcal{S}im$ that perfectly simulates honest transcripts (A, B), and user's key is set to K_1. The change being just syntactic: $\mathsf{Succ}_{\mathbf{G}_0} = \mathsf{Succ}_{\mathbf{G}_1}$.

Game $\mathbf{G}_2$: We now deal with Send-queries to the user, and namely when the adversary fakes the public information PInfo sent to the user: if the majority of at least t_r tuples $\mathsf{PInfo}' = (\Pi', \Sigma', \mathsf{SSInfo}', C')$ contains a commitment C' different from the expected commitment C, we make the user play as usual, but eventually set $K \leftarrow \perp$.

This makes a difference only if in the end this commitment would have been accepted by the user with respect to his password pw. Since we use a hash function H_3 modeled as a random oracle, C' must have been obtained with a query containing pw, or the probability to be valid is $1/2^{\ell_3}$: We thus define the event PWinC to be true if $C' \neq C$ but C' is the result of a query of H_3 on a tuple that contains pw. And at the end, after the answer b', if PWinC is set, one sets the output bit S at random instead of $(b' = b)$. In this game, we reduce the success probability of the adversary, but only when PWinC happens: $\mathsf{Succ}_{\mathbf{G}_1} \leq \mathsf{Succ}_{\mathbf{G}_2} + 1/2^{\ell_3} + \Pr[\mathsf{PWinC}]/2$. This event PWinC can be evaluated by looking at each of the queries asked to H_3 and then checking whether it contains pw or not.

Game $\mathbf{G}_3$: We continue in the same vein for fake public information PInfo′ (but correct C) sent to the user: if the majority of at least t_r tuples $\mathsf{PInfo}' = (\Pi', \Sigma', \mathsf{SSInfo}', C)$ contains public information different from the expected ones (the PInfo generated during the honest initialization phase), we make the user play as usual, but eventually set $K \leftarrow \perp$. Since the hash value C is unchanged, the input $(\mathsf{pw}, \Pi', \Sigma', \mathsf{SSInfo}')$ must be unchanged, unless one finds a collision for H_3:

$$\mathsf{Succ}_{\mathbf{G}_2} \leq \mathsf{Succ}_{\mathbf{G}_3} + q_3^2/2^{\ell_3+1}.$$

Game $\mathbf{G}_4$: We continue with the simulation of the user, but when the majority PInfo is the expected one, which guarantees the use of the correct public keys, and thus the knowledge of the associated secret keys. We now use $\mathcal{S}im_U$, that perfectly simulates a flow A from the user to a server, and can decide on the honest behavior of the server thanks to the server' secret key x_k to evaluate $\mathsf{DDH}_g(y_k, \cdot, \cdot)$. If the behaviours of at least t_r of the servers are correct, thanks to the RGTSSS scheme, the user accepts with $K \leftarrow K_1$, otherwise the user aborts with $K \leftarrow \perp$. Since the OPRF

protocol uses the random blinding factor α, either $C = H_1(\mathsf{pw})^{x_k}$ or C is a random element in $\mathbb{G}$, unless $B = 1$, hence the verification. Applying the hash function H_2 to obtain the final value R assures us that the shares are either correct, or random, hence the encoding into random non-malleable fingerprints in the RGTSSS process.

In the previous game, RGTSSS guaranteed the recovery of the secret in exactly the same cases as here: $\mathsf{Succ}_{\mathbf{G}_3} = \mathsf{Succ}_{\mathbf{G}_4}$. We remark that during this game, the Send-queries are answered without calling the H_1 oracle, neither H_2 oracle is used for the reconstruction of K. Instead, after the $\mathsf{DDH}_g(y, \cdot, \cdot)$ check (using the secret key x), the secret K is directly set to K_1 (or to $\perp$ if too many failures).

Game $\mathbf{G}_5$: Since the secret sharing reconstruction is not used anymore, we can thus replace all the shares $(s_1, \ldots, s_n)$ by random and independent values and generate SSInfo accordingly in the initialization phase. We know that until the adversary does not get more than $t_\ell = n/4$ of these shares, it cannot detect whether they are random or correct: let us define the event PWinF to be true if more than $n/4 - q_c$ queries have been asked to the H_2 oracle for the un-corrupted key y_k on pw with the correct CDH value, since the adversary can evaluate on its own the values under the q_c corrupted servers. And at the end, after the answer b', if PWinF is set, one sets S at random. As in Game $\mathbf{G}_2$, we have the upper-bound: $\mathsf{Succ}_{\mathbf{G}_4} \leq \mathsf{Succ}_{\mathbf{G}_5} + \Pr[\mathsf{PWinF}]/2$. Using the servers' secret keys $(x_k)_k$ (to test $\mathsf{DDH}_g(y_k, \cdot, \cdot)$ validity), the simulator can check whether pw has been queried with the right inputs to H_2 to learn some π_k, and how many times, to set the event PWinF.

Game $\mathbf{G}_6$: Instead of choosing the shares at random, one generates $\Sigma = (\sigma_k)_k$ and SSInfo at random, without computing the π_k's: $\mathsf{Succ}_{\mathbf{G}_5} = \mathsf{Succ}_{\mathbf{G}_6}$.

Game $\mathbf{G}_7$: We now deal with Send-queries to the servers, and replace them by the simulator $\mathcal{S}im_S$ to provide answers, using the server's secret key x_k to evaluate $\mathsf{CDH}_g(y_k, \cdot)$: $\mathsf{Succ}_{\mathbf{G}_6} = \mathsf{Succ}_{\mathbf{G}_7}$.

Game $\mathbf{G}_8$: We now replace the hash value C in the initialization phase by a random C. This is indistinguishable because of the random oracle property: $\mathsf{Succ}_{\mathbf{G}_7} = \mathsf{Succ}_{\mathbf{G}_8}$.

In this last game, one can note that the password pw is not used anymore during the simulation, but just to determine whether the events PWinC or PWinF happened to define the game output S. In addition, K_1 does not appear any more during the initialization phase (it was just used for the secret sharing, while the shares have been replaced by random shares, and in the commitment, while it has been replaced by a random hash), hence one cannot make any difference between K_0 and K_1: $\mathsf{Succ}_{\mathbf{G}_8} = 1/2$. As a consequence,

$$\mathsf{Adv}(\mathcal{A}) \leq \Pr[\mathsf{PWinC}] + \Pr[\mathsf{PWinF}] + (q_3^2 + 2) \cdot 2^{-\ell_3}.$$

We thus now have to evaluate the probabilities of the two events PWinC or PWinF to happen, which can be done by choosing a random password pw at the very end only (since it is not used anymore during the initialization phase, nor in the reconstruction): About $\Pr[\mathsf{PWinC}]$, it is clearly upper-bounded by $q_u/\#\mathcal{D}$, since q_u is the maximal number of fake commitment attempts containing the right pw that can be different from the expected ones; On the other hand, PWinF means that the adversary managed to get $n/4 - q_c$ evaluations of the PRFs under the chosen pw. But unless the adversary gets more evaluations than the number q_s of queries asked to the servers, the number of *bad* passwords (for which

he knows at least $n/4 - q_c$ evaluations of the PRFs) is less than $q_s/(n/4 - q_c)$. So the probability that the chosen pw is such a bad password is less than $q_s/(n/4 - q_c) \times 1/\#\mathcal{D}$.

The following lemma leads to $\Pr[\mathsf{PWinF}] \leq q_s/(n/4-q_c)\times 1/\#\mathcal{D}+n\times \mathsf{Succ}^{\mathsf{omgdh}}(q_1, q_s, t, n\cdot q_u + q_2)$, which concludes the proof of the theorem.

Lemma 4.4. *Unless one can break a* $(q_1, q_s) - \mathsf{OMGDH}$ *with* $(n \cdot q_u + q_2)$ *queries to the* DDH*-oracle, no adversary, that involves* q_s *instances of the servers,* q_u *instances of the user, and asks* q_1 *queries to* H_1 *and* q_2 *queries to* H_2*, can get more evaluations of the PRF than the number* q_s *of queries asked to the servers.*

If we denote q_1 the number of queries to H_1, we can also denote $(h_1, \dots h_{q_1})$ the list of the answers, which are random group elements. Let us be given a random instance $(g, y = g^x, h_1, \cdots, h_{q_1})$ of the $(q_1, q_s) - \mathsf{OMGDH}$ problem (see 4.2.2), then our simulator uses $y^* = y$ for a randomly chosen server k^*, and $y_k = g^{x_k}$ for random scalars x_k, for the other servers. Getting one-more evaluation of the PRF (under non-corrupted keys) than the number of queries to the (non-corrupted) servers means that this must be the case for at least one of the non-corrupted servers: we hope the k^*-server to be one of them. Since it is chosen at random, this is a correct guess with probability greater than $1/n$.

For the simulation of the q_s queries A to the honest servers, for the k^*-server, the simulator makes one $\mathsf{CDH}_g(y^*, \cdot)$-query, while for the others the secret key x_k is known. For the (at most $n \times q_u$) transcripts (A, B) obtained by the honest user with the adversary, the simulator makes one $\mathsf{DDH}_g(y^*, \cdot, \cdot)$-query when the adversary plays the role of the k^*-server, but can use x_k otherwise. Getting one more evaluation of F_{k^*} than the number q of queries to the k^*-server means that for at least $q + 1$ queries (pw_i, y^*, H) to the random oracle H_2, $H = \mathsf{CDH}_g(y^*, H_1(\mathsf{pw}_i))$. Since $H_1(\mathsf{pw}_i)$ has been answered by one of the h_j, one gets $q + 1$ correct values $\mathsf{CDH}_g(y^*, h_j)$, that can be detected using the $\mathsf{DDH}_g(y^*, \cdot, \cdot)$ oracle on all the q_2 inputs to H_2. We can of course upper-bound q by q_s, hence the lemma.

4.5.5 Protocol II: DDH-based PRF

Our second instantiation makes use of the Naor and Reingold [NR97] pseudorandom function. We consider the group $\mathbb{G} = \langle g \rangle$ of prime order q that is a safe prime: $q = 2s + 1$. In the multiplicative group of scalar $\mathbb{Z}_q^*$, we consider the cyclic group $\mathbb{G}_s$ of order s (this is the group of elements in $\mathbb{Z}_q^*$ with Jacobi symbol equals to $+1$). In both groups, the DDH assumption can be made.

The PRF key is a tuple $a = (a_0, a_1, \dots, a_\ell) \xleftarrow{\$} (\mathbb{G}_s \backslash \{1\})^{\ell+1}$, and $F_a(x) = g^{a_0} \prod a_i^{x_i}$, where $x = (x_1, x_2, \dots, x_\ell) \in \{0,1\}^\ell$. This function has been proven to be a PRF under the DDH assumption [NR97] on ℓ-bit inputs. It also admits a simple oblivious evaluation (just the messages C and G from Figure 4.4), using a multiplicatively homomorphic encryption scheme in $\mathbb{G}_s$, such as ElGamal for $(\mathsf{Enc}_{\mathsf{pk}}, \mathsf{Dec}_{\mathsf{sk}})$, which allows the computation of C from x, α, and the ciphertexts $\{c_i\}_i$. Unfortunately, without additional proofs, this is not secure against malicious users, since it works only for honest inputs $x \in \{0,1\}^\ell$. Hence the more involved protocol presented in Figure 4.4 that makes use of a zero-knowledge proof of knowledge of $(x_i)_i \in \{0,1\}^\ell$ and $\alpha \in \mathbb{G}_s$. This can be efficiently done under the sole DDH assumption. Whereas our oblivious evaluation of the PRF is in the standard model, overall, the PPSS protocol based on this OPRF is in the random oracle model as it makes use of the RGTSSS. As a consequence, one could replace the interactive ZK proofs by NIZK proofs "à la Schnorr". This would reduce the number of flows to only 2.

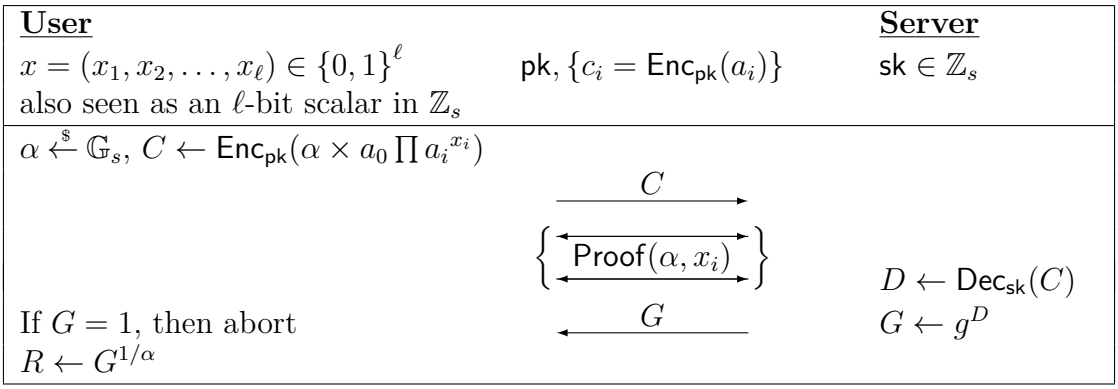


Figure 4.4: Secure Oblivious Evaluation of the NR-PRF

4.5.6 Security Proof of the Protocol II

Theorem 4.5. *For any adversary $\mathcal{A}$, against the Protocol II using both ElGamal and Cramer-Shoup encryption schemes, that corrupts no more than q_c servers, involves at most q_s instances of the servers, and q_u instances of the user*

$$\mathsf{Adv}(\mathcal{A}) \leq \left(q_u + \frac{4q_s}{n - 4q_c}\right) \times \frac{1}{\#\mathcal{D}} + ((n - q_c)\ell + 4) \times \mathsf{Adv}^{\mathsf{ddh}}(t + q_s t_{\mathsf{exp}}) + 3 \times \mathsf{Succ}^{\mathsf{2nd}}_{\mathcal{H}}(t) + 6q_u/q,$$

where ℓ is the size of the password and t_{exp} the time for an exponentiation.

The proof will be performed by a sequence of games, as for the previous protocol, focusing on a unique user, against a static adversary (the corrupted servers are known from the beginning). A change from the general description of the PPSS protocol in this particular case consists a more efficient way to compute the commitment $C = \mathsf{Commit}(\mathsf{pw}, \Pi, \Sigma, \mathsf{SSInfo}, K; r)$, by first a fingerprint $H = \mathcal{H}(\Pi, \Sigma, \mathsf{SSInfo}, K)$ with a second-preimage-resistant hash function $\mathcal{H}$, and then $C = \mathsf{Enc}(\mathsf{pw}, H; r)$, with an $\mathsf{IND-CCA}$ encryption scheme. This improves the efficiency, as the information $(\Pi, \Sigma, \mathsf{SSInfo}, K)$ may be long. More precisely, we use the Cramer-Shoup encryption scheme, denoted $(\mathsf{CS.Enc}, \mathsf{CS.Dec})$, for the extractable commitment. We will use the simulators presented in Figure 4.6, where the c_i's have been replaced by random ciphertexts, which is indistinguishable under the $\mathsf{IND-CPA}$ security level of the ElGamal encryption scheme, denoted $(\mathsf{EG.Enc}, \mathsf{EG.Dec})$. $\mathcal{S}im_U$ knows the encrypted value D, and can thus check the answer. In addition, using Proof, a zero-knowledge proof of knowledge of $(x_i)_i \in \{0,1\}^\ell$, $\alpha \in \mathbb{G}_s$, and additional random coins such that C is correct, we show that this enhanced protocol can check the correctness of the answers from the server. Indeed, from the extractor of Proof, $\mathcal{S}im_S$ can extract $(x_i)_i \in \{0,1\}^\ell$ to ask the PRF oracle, that answers either correctly or at random, as well as $\alpha \in \mathbb{G}_s$, to send a blinded answer to the client.

More precisely, using ElGamal encryption, we have (using component-wise multiplication)

$$C = (g^r, h^r\alpha) \times c_0 \prod c_i^{x_i} = (g^r, h^r) \times c_0 \prod c_i^{x_i} \times (1, \alpha),$$

and one has to prove its knowledge of $(x_i)_i \in \{0,1\}^\ell$, $\alpha \in \mathbb{G}_s$, and $r \in \mathbb{Z}_s$ that satisfy this relation. To get a straightline extraction, one can use a Cramer-Shoup encryption of α and u^x, for a generator $u \in \mathbb{G}_s$ (assuming the use of a password small enough to allow discrete

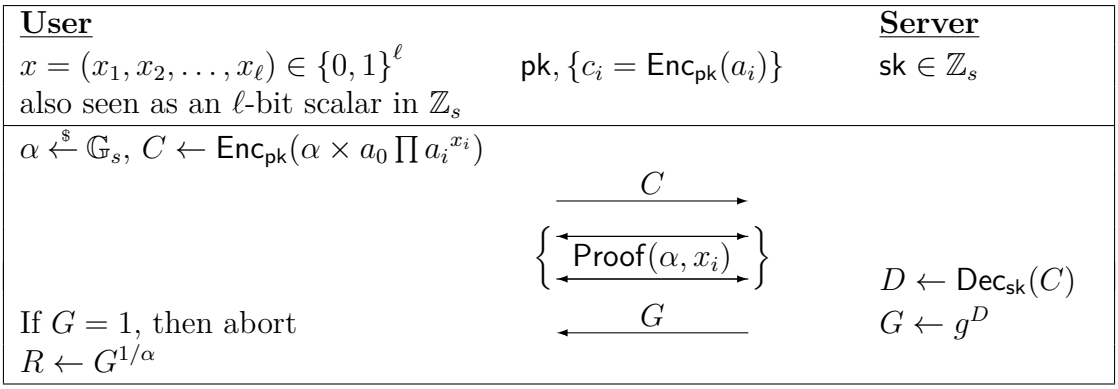
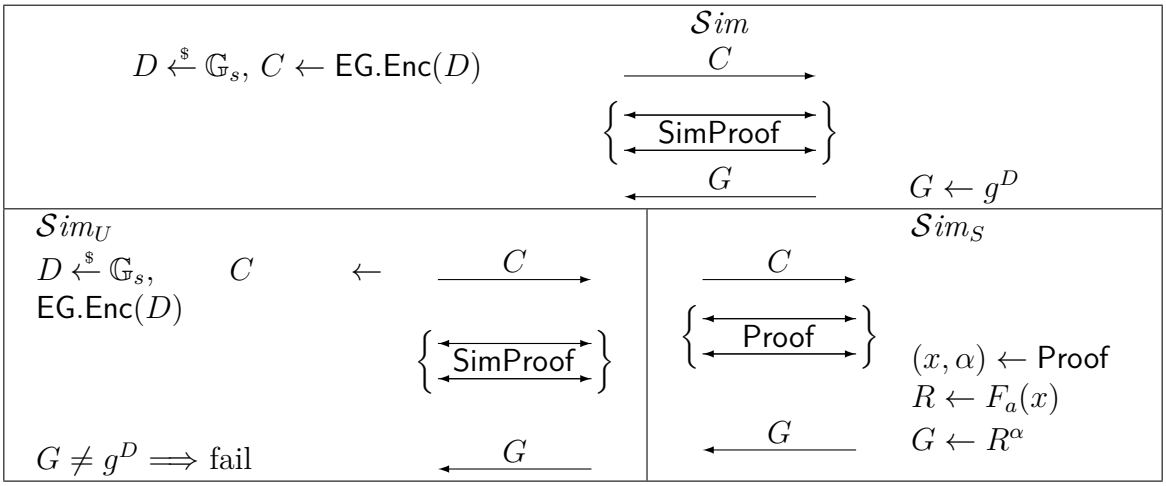


Figure 4.5: Secure Oblivious Evaluation of the NR-PRF



$\mathcal{S}im_S$ extracts x from Proof to build R, the expected PRF value. The ciphertexts c_i have been replaced by random encryptions.

Figure 4.6: Simulators for the OPRF based on CDH

logarithm computation), where the latter can be seen as $u^x = \prod (u^{2^i})^{x_i}$. Otherwise, one can encrypt u^{x_i} for each index i.

One should note that we only have to do proofs in $\mathbb{G}_s$, which are classical Schnorr-like proofs.

Game $\mathbf{G_0}$: This initial game corresponds to the real attack game. As in the proof for the PPSS Protocol I, we emulate the initialization phase, which is honestly performed: we choose one random pw, n random keys $(\mathsf{sk}_k)_k$ for the ElGamal encryption scheme and the PRF keys $a_k = (a_{k,0}, \ldots a_{k,\ell})_k$ that represent the servers' secret information. We generate their corresponding public part $\Pi = (\mathsf{pk}_k, (c_i = \mathsf{Enc}_{\mathsf{pk}_k}(a_{k,i}))_k)$, and one random value corresponding to the secret K, together with a secret sharing $(s_1, \ldots, s_n, \mathsf{SSInfo})$ of $R = K \| r$, for a random r. These shares are masked using the values $(\pi_k)_k$, obtained as PRF evaluations of pw under all the secret strings a_k of the servers. We then set $\Sigma = (\sigma_k = \pi_k \oplus s_k)_k$.

The same random value r in $R = K \| r$ is used to compute the commitment $C = \mathsf{CS.Enc}(\mathsf{pw}, H; r)$, where $H = \mathcal{H}(\Pi, \Sigma, \mathsf{SSInfo}, K)$.

One also chooses a second random key K_0, as well as a bit $b \xleftarrow{\$} \{0,1\}$, both used in Test-queries: if a key K_1 is reconstructed, the Test-query outputs K_b, if the reconstruction is not completed or failed, the Test-query outputs UNDEFINED or $\perp$. For the reconstruction, we simulate all the instances, the user and the servers, in Execute and Send-queries, as the real players would do.

The adversary eventually outputs its guess b' for the bit b. The output of the game is the success bit $S = (b' = b)$. By definition we have :

$$\mathsf{Succ}_{\mathbf{G}_0} = \Pr[S] \qquad\qquad \mathsf{Adv}(\mathcal{A}) = 2 \times \mathsf{Succ}_{\mathbf{G}_0} - 1$$

Game $\mathbf{G}_1$: We first deal with Execute-queries, without modifying the initialization. We replace the user and the servers in the reconstruction protocol by the simulator $\mathcal{S}im$ from figure 4.6. This perfectly simulates honest transcripts (C, Proof, G), and user's key is set to K_1. The change for the values C and G is just syntactic, the two values are equivalent to the real ones:

$$\mathsf{Succ}_{\mathbf{G}_0}(\mathcal{A}) = \mathsf{Succ}_{\mathbf{G}_1}(\mathcal{A}).$$

Game $\mathbf{G}_2$: We consider an adversary that fakes the public information PInfo in Send-queries to the user: if the majority of at least t_r tuples $\mathsf{PInfo}' = (\Pi', \Sigma', \mathsf{SSInfo}', C')$ contains a commitment C' different from the expected commitment C, we make the user play as usual, but eventually set $K \leftarrow \perp$.

The reconstruction protocol guarantees that if the majority tuple $\mathsf{PInfo} = (\Pi, \Sigma, \mathsf{SSInfo}, C)$ does not contain the expected commitment C, the user aborts with $K \leftarrow \perp$. This makes a difference only if in the end this commitment would have been accepted by the user with respect to his password. Since we use a perfectly binding commitment (an encryption scheme), the ciphertext C' must contain the correct pw: We thus define the event PWinC to be true if $C' \neq C$ but contains pw.

We simulate the game by checking if PWinC is set at the end, after receiving the answer b'. If so, one sets the output bit S at random instead of $(b' = b)$. In this game, we reduce the success probability of the adversary, but only when PWinC happens:

$$\mathsf{Succ}_{\mathbf{G}_1} \leq \mathsf{Succ}_{\mathbf{G}_2} + \Pr[\mathsf{PWinC}]/2$$

This event PWinC can be evaluated by decrypting the commitment $C = \mathsf{CS.Enc}(\mathsf{pw}, H; r)$ using the decryption key, and then checking whether it contains pw or not.

Game $\mathbf{G}_3$: We continue in the same vein for fake public information PInfo' (but correct C) sent to the user: if the majority of at least t_r tuples $\mathsf{PInfo}' = (\Pi', \Sigma', \mathsf{SSInfo}', C)$ contains public information different from the expected ones (the PInfo generated during the honest initialization phase), we make the user play as usual, but eventually set $K \leftarrow \perp$.

The RGTSSS protocol generates either correct fingerprints when computed from shares obtained with the help of honest servers or random independent fingerprints when the servers cheat and the shares obtained by the client are not the expected ones. The reconstruction using RGTSSS guarantees the recovery of the secret in the honest case. When the value C is unchanged, the value $H' = \mathcal{H}(\Pi', \Sigma', \mathsf{SSInfo}', K')$ must be the same as the initial commitment input $H = \mathcal{H}(\Pi, \Sigma, \mathsf{SSInfo}, K)$ in order to

be accepted by the user, since this is an encryption scheme, with a unique decryption. If in the end of the previous the simulator was accepting the key K', this means that we have a second pre-image $(\Pi', \Sigma', \mathsf{SSInfo}', K')$ of the initial $H = \mathcal{H}(\Pi, \Sigma, \mathsf{SSInfo}, K)$. As a consequence, this simulation is perfectly indistinguishable from the previous one unless one finds a second pre-image to $H = \mathcal{H}(\Pi, \Sigma, \mathsf{SSInfo}, K)$ (where t is essentially the running time of $\mathcal{A}$):

$$\mathsf{Succ}_{\mathbf{G}_2}(\mathcal{A}) \leq \mathsf{Succ}_{\mathbf{G}_3}(\mathcal{A}) + \mathsf{Succ}^{\mathsf{2nd}}_{\mathcal{H}}(t).$$

Game $\mathbf{G}_4$: We are still dealing with Send-queries to the user, but we consider the case of the event $\neg\mathsf{PWinC}$. We now use $\mathcal{S}im_U$, that perfectly simulates a flow C from the user to a server, and can decide on the honest behavior of the server by choosing itself the value D (the decryption of C), and using it to check the correctness of servers' answers.

If the behaviors of at least t_r of the servers are correct, the user accepts with $K \leftarrow K_1$, otherwise the user aborts with $K \leftarrow \perp$.

In the previous game, the RGTSSS guaranteed the recovery of the secret in exactly the same cases as here:

$$\mathsf{Succ}_{\mathbf{G}_3} = \mathsf{Succ}_{\mathbf{G}_4}.$$

We remark that during this game, the Send-queries are answered without computing the PRF for the reconstruction of K. Instead, after $G = g^D$ check, the secret K is directly set to K_1 (or to $\perp$ if too many failures).

Game $\mathbf{G}_5$: We now deal with Send-queries to the servers, and replace them by the simulator $\mathcal{S}im_S$ to provide answers, getting x and the blinding factor α from the extractor of the proof: in order to set the appropriate output to R (that is $F_a(x)$), the server can simply answer $G \leftarrow R^\alpha$. The simulation is perfect:

$$\mathsf{Succ}_{\mathbf{G}_4} = \mathsf{Succ}_{\mathbf{G}_5}.$$

Game $\mathbf{G}_6$: In the servers' simulation, the value R is now chosen at random for new x (for the uncorrupted servers). This corresponds to replace F_a by a truly random function when calling to the PRF oracle. Under the pseudo-randomness of the Naor-Reingold PRF:

$$\mathsf{Succ}_{\mathbf{G}_5} \leq \mathsf{Succ}_{\mathbf{G}_6} + (n - q_c)\ell \times \mathsf{Adv}^{\mathsf{ddh}}(t + q_s t_{\mathsf{exp}}),$$

where t_{exp} is the additional time for exponentiations in the reduction of the PRF.

Game $\mathbf{G}_7$: Instead of choosing the π_k at random, one generates $\Sigma = (\sigma_k)_k$ and SSInfo at random. This leads to random and independent shares $(s_1, \ldots, s_n)$. We know that until the adversary does not get more than $t_\ell = n/4$ of these shares, it cannot detect whether they are independent or redundant (as should be a secret sharing): let us define the event PWinF', a little bit different from the previous proof, to be true if more than $n/4 - q_c$ queries have been asked to the servers on pw, since the adversary can evaluate on its own the values under the q_c corrupted servers. Then, from these values it could remark inconsistencies. At the end, after the answer b', if PWinF' is set, one sets S at random. Since the PRF's are replaced by truly random functions, these queries do not reveal anything on the other values of the functions, we have the upper-bound: $\mathsf{Succ}_{\mathbf{G}_6} \leq \mathsf{Succ}_{\mathbf{G}_7} + \Pr[\mathsf{PWinF}']/2$. Thanks to the extractability of x from the proof, we are able to check whether pw has been used, and how many times in order to set the event PWinF'.

Game $\mathbf{G}_8$: We now replace the commitment C in the initialization phase by a dummy commitment to 0. This is indistinguishable under the indistinguishability of the encryption scheme (Cramer-Shoup encryption), but the decryption key is required to evaluate PWinC:

$$\mathsf{Succ}_{\mathbf{G}_7} \leq \mathsf{Succ}_{\mathbf{G}_8} + \mathsf{Adv}^{\mathsf{ind-cca}}_{\mathsf{CS}}(t).$$

.

Game $\mathbf{G}_9$: The key K_1 does not appear any more in the simulation of the secret sharing, as the values PInfo have been replaced by random and independent values instead of shares and the commitment C is currently computed for 0. Then, we can replace K_1 by K_0 in the reconstruction phase, which makes the real and random cases indistinguishable:

$$\mathsf{Succ}_{\mathbf{G}_8}(\mathcal{A}) = \mathsf{Succ}_{\mathbf{G}_9}(\mathcal{A}) = 1/2.$$

In this final game, the password does not appear any more in the initialization of PInfo, and the simulator does not make use of it either, except to abort if PWinC or PWinF happen. But these events can be evaluated at the very end only, by choosing a random password pw when the adversary outputs its guess b':

- $\Pr[\mathsf{PWinC}]$ is clearly upper-bounded by $q_u/\#\mathcal{D}$, since q_u is the maximal number of fake commitment attempts that could be different from the expected one but with pw;
- $\Pr[\mathsf{PWinF}']$ is clearly upper-bounded by $q_s/(n/4 - q_c) \times 1/\#\mathcal{D}$, since $q_s/(n/4 - q_c)$ is the maximal number of passwords for which the adversary asked for $n/4 - q_c$ OPRF evaluations.

In conclusion, we have:

$$\mathsf{Adv}(\mathcal{A}) \leq \left(q_u + \frac{4q_s}{n - 4q_c}\right) \times \frac{1}{\#\mathcal{D}} + (n - q_c)\ell \times \mathsf{Adv}^{\mathsf{ddh}}(t) + 2 \times \mathsf{Adv}^{\mathsf{ind-cca}}_{\mathsf{CS}}(t) + \mathsf{Succ}^{\mathsf{2nd}}_{\mathcal{H}}(t).$$

Since we use the Crame-Shoup encryption scheme, this leads to the bound of the Theorem.

4.6 Comparisons

We can assume that PInfo is stored in the Cloud, it does not need to be sent by each server, then the global communication is linear in n. More precisely, our first protocol is quite similar to the one from [JKKX16]. Of course, we did not provide any security result in the UC framework [Can01], but our ultimate goal was the same as [JKK14]: an efficient robust password-protected secret sharing scheme, in a BPR-like security model [BPR00]. To this aim, there is no reason to use UC-secure building blocks, but tailored primitives.

Our algebraic OPRF structure is more efficient than the one in [FIPR05], since their construction makes use of Oblivious Transfers (OT) and expensive public-key operations. In the online setting, this kind of protocols are almost infeasible, as the number of desired OTs is not known in advance while our zero-knowledge proofs are much simpler to use. Given the work of Ishai *et al.* [IKNP03], a better efficiency can be achieved, considering each OT evaluation at the cost of a private-key operation. In our case, the main cost in communication is that of a single zero-knowledge proof.

Our second protocol, based on this oblivious evaluation and with an additionally CRS turns out to be much more efficient than the one from [JKK14]. Even if it uses the same Naor-Reingold PRF, the oblivious evaluation is much more efficient and relies on the DDH assumption only. Our full construction only makes use of ElGamal and Cramer-Shoup encryption schemes, and no Paillier's encryption [Pai99] nor Cramer-Shoup signature [CS99] that require both stronger assumptions, such as the strong-RSA assumption and the decisional composite residuosity assumption, and much larger parameters, which lead to huge communication load. The main reason comes from the relaxation on the OPRF: since we do not need verifiability of server's computations, it does not have to make any zero-knowledge proof, which allows us to use a much more efficient OPRF.

Chapter 5

Storing in the Cloud: Searchable Encryption

5.1 Introduction

Symmetric Searchable Encryption (SSE) schemes allow a client to encrypt a database, such as a mailbox or a relational database, in a way that enables him to efficiently search his data using keywords or attribute-value pairs. In the following, we will assume free-text documents and keyword and in experiments we will take the set of all Wikipedia pages in english. On the server, the database is encrypted and the client can later search for all documents containing some keywords or more complex boolean queries involving many keywords. Instead of storing the documents, indices corresponding to the documents are used and the client knows the mapping between them. In the seminal work by Song *et al.* [SWP00], two solutions have been considered: the first one is linear in the size of the database as the server sequentially scans it for every search query. Such solutions are considered today as inefficient, since the goal is to have search complexity, on the server side, proportional to the output size while having a large number of keyword/document pairs in the database. The second solution is based on an inverted index: for each keyword w the list of all documents matching w is associated, and was initially presented as inefficient in the dynamic setting, *i.e.* when the clients can add/remove documents.

Dynamic SSE. Today, most efficient techniques are built on this last idea and allow efficient search and update [Goh03, CM05, CJJ+13, SPS14, CGKO06, GSW04].
When we want to build secure dynamic scheme, further security issues may be considered, such as forward security: if the user looks for a keyword w and later adds a new document containing w, the server will learn that this new document has a keyword that has been looked for in the past. Practical solutions have been proposed to address forward security in [SPS14].

Multiple-Keyword Search. In 2013, Cash *et al.* [CJJ+13] proposed an efficient construction for searching *conjunctive boolean queries* of the form $\phi(w_1, \ldots, w_m)$, where ϕ is a boolean formula. More precisely, [CJJ+13] considers queries of the form $w_1 \wedge \psi(w_2, \ldots, w_m)$: a single keyword search (SKS) scheme is used to find the documents matching the first term w_1, and these results are then filtered according to the other keywords and the formula. Their Oblivious Cross Tag (OXT) protocol is a single-roundtrip protocol: the client sends

tokens derivated from the other keywords for the server to compute a Diffie-Hellman-based oblivious PRF whose results are used for lookups in a Bloom filter.

Verifiable SSE. An other security property SSE schemes try to achieve is *verifiability*. Until the work of Kurosawa and Ohtaki [KO12], servers were usually considered as *honest-but-curious* adversaries and never tried to deviate from the prescribed protocol. In the malicious adversarial model, servers could send only a small part of the answers or even non matching documents. Verifiable SSE schemes solve this issue by ensuring soundness of the search queries. Bost *et al.* [BFP16] exhibited a logarithmic lower bound of the running time of such verifiable schemes, and described a black-box construction, GSV (for Generic SSE Verification) matching this lower bound, that we will efficiently instantiate in this work. [BFP16] also presents a forward secure and verifiable scheme, derived from the forward secure construction in [SPS14].
Fisc *et al.* [FVK+15] studied the problem of malicious clients in a three party setting, involving a server, a data owner, and a client who tries to learn more than what he is allowed to.

Security Issues. One important issue is the security of these schemes. The server could act as an adversary that tries to learn information about the database and about the queries. In the security model, first formally described by Curtmola *et al.* [CGKO06], we consider a leakage function $\mathcal{L}$ that quantifies the information leaking about those from the protocol. Usually, the results sets size and the repetition of queries leak.
Cash *et al.* [CGPR15] extended the work on statistical attack of Islam *et al.* in [IKK12] by studying practical and theoretical SSE schemes, and their leakage functions. They show that a clever adversary can easily recover the queries or the database from a too leaky 'secure' SSE scheme. For example, they recommend to pad the inverted indices answers to hide the number of documents matching an encrypted query – it can be used to retrieve the clear query as keywords almost all have different numbers of matching documents.

5.2 Preliminaries

In this work, the security parameter will be denoted λ. We will also use the standard definitions of pseudo-random functions (PRF) and semantically secure symmetric encryption schemes [Gol04]. For the sake of simplicity, we suppose that the keys are strings of λ bits, and that the key generation algorithm uniformly chooses a key in $\{0,1\}^\lambda$. Unless otherwise specified, we always consider probabilistic algorithms/protocols running in time polynomial in λ, also called *efficient*. Adversaries are probabilistic polynomial time (ppt) algorithms and $\mathrm{negl}(\lambda)$ denotes a negligible function in λ.

5.2.1 SSE: Symmetric Searchable Encryption

A *database* $\mathsf{DB} = (\mathsf{ind}_i, \mathsf{W}_i)_{i=1}^d$ is a tuple of index/keyword-set pairs with $\mathsf{ind}_i \in \{0,1\}^\lambda$ and $\mathsf{W}_i \subseteq \{0,1\}^*$. The set of keywords of the database DB is $\mathsf{W} = \cup_{i=1}^d \mathsf{W}_i$. We set $m = |\mathsf{W}|$ to be the total number of keywords in DB, and $N = \sum_{i=1}^d |\mathsf{W}_i|$ to be the number of document/keyword pairs (as already said, we identify documents with their indices). We denote by $\mathsf{DB}(w)$ the set of documents containing keyword w, *i.e.* $\mathsf{DB}(w) = \{\mathsf{ind}_i | w \in \mathsf{W}_i\}$. N can also be written as $N = \sum_{w \in \mathsf{W}} |\mathsf{DB}(w)|$.

A *dynamic searchable encryption scheme* $\Pi = (\mathsf{Setup}, \mathsf{Search}, \mathsf{Update})$ consists of one algorithm and two protocols between a client and a server:

- $\mathsf{Setup}(\mathsf{DB})$ is an algorithm that takes as input a database DB. It outputs a pair $(\mathsf{EDB}, K, \sigma)$ where K is a secret key, EDB the encrypted database, and σ the client's state.
- $\mathsf{Search}(K, \sigma, q; \mathsf{EDB}) = (\mathsf{Search}_C(K, \sigma, q), \mathsf{Search}_S(\mathsf{EDB}))$ is a protocol between the client with input the key K, its state σ, and a search query q, and the server with input EDB. We write $(V, \sigma', \tau) \xleftarrow{\$} \mathsf{Search}(K, \sigma, q; \mathsf{EDB})$ to mean that V, σ' and τ are sampled by running the protocol with these inputs, V being the client output, σ' the new state of the client, τ the messages sent by the client (the transcript). V (and σ') can take the special value `REJECT`. For schemes supporting boolean queries, a search query consists of a boolean formula ϕ and a set of keywords $(w_1, \ldots, w_n)$. For SKS schemes, a search query is restricted to a unique keyword w.
- $\mathsf{Update}(K, \sigma, \mathsf{op}, \mathsf{in}; \mathsf{EDB}) = (\mathsf{Update}_C(K, \sigma, \mathsf{op}, \mathsf{in}), \mathsf{Update}_S(\mathsf{EDB}))$ is a protocol between the client with input the key K and state σ, an operation op and an input in parsed as the index ind and a set W of keywords, and the server with input EDB. As in previous papers [CJJ+14], the update operations are taken from the set $\{\mathsf{add}, \mathsf{edit}^+, \mathsf{del}, \mathsf{edit}^-\}$, meaning, respectively, the addition of a full document, of a document/keyword pair, the deletion of a full document and the deletion of a single pair. We write $(\sigma', \tau; \mathsf{EDB}') \xleftarrow{\$} \mathsf{Update}(K, \sigma, \mathsf{op}, \mathsf{in}; \mathsf{EDB})$ to mean that σ', τ and EDB' are sampled by running the protocol, σ' being the output of the client *i.e.* its new state, τ being the client's transcript and EDB' be the server output *i.e.* the updated encrypted database. Note that σ' can take the special value `REJECT`.

5.2.2 Correctness.

The correctness of an SSE scheme is the basic property we want to ensure: the search protocol must return the correct result for every query, except with negligible probability. We formally define correctness in 5.4.1.

5.2.3 Confidentiality.

The confidentiality definition of an SSE scheme uses the real world vs. ideal world formalization [CGKO06]. It is parametrized by a *leakage function* $\mathcal{L} = (\mathcal{L}^{\mathsf{Stp}}, \mathcal{L}^{\mathsf{Srch}}, \mathcal{L}^{\mathsf{Updt}})$ describing what the protocol leaks to the adversary, and formalized as a stateful algorithm. The definition ensures that the scheme does not reveal any information beyond the ones it can infer from the leakage function.

More precisely, the adversary A chooses a database DB, and is given back EDB generated using $\mathsf{Setup}(\mathsf{DB})$ in the real case, or $S(\mathcal{L}^{\mathsf{Stp}}(\mathsf{DB}))$ in the ideal case, where S is a *simulator*, an algorithm which will try to mimic the real scheme using only the leakage. Then, he repeatedly performs search and update queries with an input in and receives the transcripts generated running the $\mathsf{Search}(\mathsf{in})$ (resp. $\mathsf{Update}(\mathsf{in})$) protocol in the real case, or the simulator $S(\mathcal{L}^{\mathsf{Srch}}(\mathsf{in}))$ (resp. $S(\mathcal{L}^{\mathsf{Updt}}(\mathsf{in}))$) in the ideal case. Eventually, A outputs a bit, which is used by the game as its own output.

We say that the scheme is $\mathcal{L}$-*adaptively-secure* if for all adversary A, there exists an efficient simulator S such that the adversary A cannot distinguish the real world from the ideal world with non-negligible probability. We formally define this notion in 5.4.3.

5.2.4 Soundness.

The soundness of a scheme ensures that, if the server tries to deviate from the protocol, he will get caught. In particular, it implies that the server cannot cheat by returning erroneous matches to a search query. The formal definition (Appendix 5.4.2) introduced in [BFP16] closely follows the soundness definition of interactive provers [Gol04]. In particular, it not only ensures that all the documents retrieved by the server are matching the search query, but also that all the matching results have been given to the client (not only a subset). It also protects the client against replay attacks from the server (*e.g.* re-sending results of a search query, that have been later removed by an update).

An adversary would win the soundness game if he is able to forge a search-query response, validated by the client, but that encodes a different result than the real result. The client has to *verify* the results the server sent.

The soundness requirement is slightly different from the UC secure definition of Kurosawa and Ohtaki [KO12], but Theorem 1 of [KO12] shows that we get UC security from confidentiality and soundness.

Query Pattern. In almost all SSE works, because of their deterministic nature, the schemes leak the repetition of tokens sent by the client to the server. For example, two search-queries using the same keywords will leak the repetition of these keywords. In previous works [CGKO06, CJJ+13], this is called the *search pattern*. In other constructions (*e.g.* [CJJ+14]), this type of leakage is not limited to search queries: repetition of keywords for both search and update queries leaks. Hence, we call it the *query pattern*. The query pattern $\overline{x}$ of a token x is constructed by the following algorithm qp: initially qp creates an empty list L as state and sets a counter $i \leftarrow 0$, and then on input x, it increments i, adds (i, x) to L and outputs $\overline{x} = \{j | (j, x) \in L\}$. Said otherwise, $\overline{x}$ returns the timestamps of all the previous queries in which x was used.

In this work, we will also use the notation $\mathsf{HistDB}(w)$: it is the set of documents historically added to DB matching keyword w. In particular, it includes documents that have been added and later deleted.

5.2.5 Notations and Tools

5.2.5.1 Games.

Our security and correctness notions are defined using code-based games [BR06]. A game G is a set of oracle procedures – including an initialization Init procedure and a finalization Final procedure – that is executed with an adversary A, *i.e.* A has access to the procedures, with some possible restrictions. For instance, the Init oracle is always the first one to be called and Final the last one, once A halted, taking A's output as input. The output of Final is called the output of the game and is denoted $G^A(\lambda)$. When Final is omitted, it just forwards the adversary's output.

5.2.5.2 Protocols.

In the paper, we will construct and use some two-party protocols, involving a client C and a server S. We will denote a protocol P as

$$P(\mathtt{input}_C; \mathtt{input}_S) = (P_C(\mathtt{input}_C), P_S(\mathtt{input}_S))$$

meaning that P_C (resp. P_S) is executed by the client (resp. the server) with input $\texttt{input}_C$ (resp. $\texttt{input}_S$). We write

$$(\texttt{out}_C; \texttt{out}_S) \xleftarrow{\$} C(\texttt{input}_C) \leftrightarrow S(\texttt{input}_S)$$

to mean that $\texttt{out}_C$ and $\texttt{out}_S$ are the outputs of the interaction between C on input $\texttt{input}_C$ and S on input $\texttt{input}_S$. When C and S run a protocol P, we simply write $(\texttt{out}_C; \texttt{out}_S) \xleftarrow{\$} P(\texttt{input}_C; \texttt{input}_S)$. In the following, we will often consider the messages τ sent by the client (the transcript) and include them to the client's output.

5.3 Building Blocks

In this work, we build two primitives. We present these primitives, their syntax, their functionalities, and their security requirements.

5.3.1 Verifiable Single-Keyword SSE

The first component we use in our construction is a verifiable SSE scheme supporting single-keyword queries only. Security definitions are the same as for generic SSE but with queries restricted to one keyword only. Such schemes have been studied by Kurosawa and Ohtaki [KO12, KO13] and Bost *et al.* [BFP16]. Kurosawa and Ohtaki described successively static and dynamic constructions in their papers. In [BFP16], the authors propose two constructions of verifiable SSE: the first one, GSV is a generic construction based on a non-verifiable SSE scheme, a verifiable hash table and a set hash function; the second one, Verif-SPS is a modification (mostly on the client side) of the scheme of Sefanov *et al.* [SPS14].

5.3.2 Encrypted Sets (E-Set)

E-Sets, or *encrypted sets*, instantiate secure outsourced sets. They encode a set in a way that allows confidential and verifiable membership queries, and supports updates.

Syntax and Security. An E-Set instantiation $\chi = (\mathsf{Setup}, \mathsf{Retrieve}, \mathsf{Update})$ consists of one algorithm and two protocols.

- $\mathsf{Setup}(X)$ is an algorithm that takes as input a subset X of a universe set $\mathcal{X}$ and outputs the tuple $(K_E, \sigma_E, \mathsf{ESet})$, where K_E is a secret key, σ_E is the client's state, and ESet the encrypted set.
- $\mathsf{Retrieve}(K_E, \sigma_E, x; \mathsf{ESet})$ is a protocol between the client with input the key K_E, its state σ_E, an element $x \in \mathcal{X}$ and the server with input ESet. The notation $(b, \tau) \xleftarrow{\$} \mathsf{Retrieve}(K_E, \sigma_E, x; \mathsf{ESet})$ means that b and τ are sampled by running the protocol with these inputs, b being the client's output and τ the messages sent by the client. Nominally, $\mathsf{Retrieve}$ should output the bit $(x \in X)$. b can take the special value `REJECT`.
- $\mathsf{Update}(K_E, \sigma_E, x, \mathsf{op}; \mathsf{ESet})$ is a protocol between the client with input the key K_E, its state σ_E, an element $x \in \mathcal{X}$, an operation $\mathsf{op} \in \{\mathsf{add}, \mathsf{del}\}$ and the server with input ESet. $(\sigma'_E, \tau; \mathsf{ESet}') \xleftarrow{\$} \mathsf{Update}(K_E, \sigma_E, x, \mathsf{op}; \mathsf{ESet})$ means that τ and ESet' are sampled

by running the protocol with these inputs, τ being the messages sent by the client, σ'_E the new client state and ESet' the server output.

We require E-Set to have similar security properties to SSE: correctness, confidentiality and soundness. E-Sets should indeed hide as much as possible about the set X and the queried elements x, including the results of membership queries, and it should be infeasible for an adversary to make the client accept an incorrect answer of such a query. To define these notions we use security games – derived from the ones of SSE – described in 5.4.4. Informally, we say that χ is *correct* if the $\mathsf{Retrieve}$ protocol returns the correct results, *i.e.* when run on input x, it returns 1 if $x \in X$, 0 otherwise, except with negligible probability. For the confidentiality, or privacy, as for SSE, the definition is parametrized by a leakage function $\mathcal{L}_E = (\mathcal{L}_E^{\mathsf{Stp}}, \mathcal{L}_E^{\mathsf{Ret}}, \mathcal{L}_E^{\mathsf{Updt}})$ and uses two games $\text{ESetReal}_A^{\Sigma}$ and $\text{ESetIdeal}_{A,S,\mathcal{L}_E}^{\Sigma}$. In both games, the adversary chooses a set X and receives ESet, and then repeatedly and adaptively issues Update or Search queries which respectively return transcripts from the Update and $\mathsf{Retrieve}$ protocols. In ESetReal the responses are generated using the real protocols, while in ESetIdeal, they come from a simulator S using the informations of the leakage function $\mathcal{L}_E$. The two views should be indistinguishable to any adversary. Finally, for soundness, we follow the same ideas as for SSE, with an ESetSound game in which the adversary wins if it manages to make the client accept an incorrect answer.

5.3.3 Content-Hiding E-Sets.

As mentioned earlier, we hope for an E-Set instantiation to be somewhat hiding about the content of the set, and in particular about its elements. Something we minimally want is the leakage function $\mathcal{L}_E$ to be of the form

$$\mathcal{L}_E^{\mathsf{Stp}}(X) = \mathcal{L}_E'^{\mathsf{Stp}}(|X|), \mathcal{L}_E^{\mathsf{Ret}}(x) = \mathcal{L}_E'^{\mathsf{Ret}}(\overline{x}, b), \text{ and } \mathcal{L}_E^{\mathsf{Updt}}(\mathsf{op}, x) = \mathcal{L}_E'^{\mathsf{Updt}}(\mathsf{op}, \overline{x}),$$

where $\mathcal{L}'^{\mathsf{Stp}}$, $\mathcal{L}_E'^{\mathsf{Ret}}$, $\mathcal{L}_E'^{\mathsf{Updt}}$ are stateful algorithms, and $\overline{x}$ is the query pattern defined in Section 5.2.4, indicating which queries have an identical query element x, and $b = 1$ iff $x \in X$. In this case, the E-Set leaks, at most, the size of the set, the query repetitions, and their results.

It is easy to show that, without loss of generality, we can always suppose that an instantiation χ satisfies this property: we can easily transform any instantiation χ in a content-hiding instantiation $\widetilde{\chi}$, given a pseudo-random permutation (PRP) P over $\mathcal{X}$, just by applying P to the elements of $\mathcal{X}$. Hence, in the rest of the paper, the E-Sets leakage functions will be written with the reduced form.

5.4 SSE Security Definitions

We give here the formalized definitions of security for SSE schemes, using security games. The games use the function Apply that outputs DB updated according to the input operation op, and the input in for that operation.

5.4.1 Correctness

Definition 5.1 (SSE Correctness)**.** *An SSE scheme* Π *is* correct *if for all adversary* A, $\mathsf{AdvCor}_A^{\mathsf{SSE},\Pi}(\lambda)$ *is negligible in* λ*, where*

$$\mathsf{AdvCor}_A^{\mathsf{SSE},\Pi}(\lambda) = \Pr[\text{SSECorr}_{\Pi}^{A}(\lambda) = 1].$$

$\underline{\mathsf{Init}(\mathsf{DB})}$
$(\mathsf{EDB}, K, \sigma) \leftarrow \mathsf{Setup}(\mathsf{DB})$

return EDB

$\underline{\mathsf{Search}(q)}$
$(V, \sigma, \tau) \xleftarrow{\$} \mathsf{Search}(K, \sigma, q; \mathsf{EDB})$
if $V \neq \mathsf{DB}(q)$ **or** $\sigma = \texttt{REJECT}$
$\texttt{win} \leftarrow \texttt{true}$

return τ

$\underline{\mathsf{Update}(\mathsf{op}, \mathsf{in})}$
$(\sigma', \tau; \mathsf{EDB}') \xleftarrow{\$}$
$\mathsf{Update}(K, \sigma, \mathsf{op}, \mathsf{in}; \mathsf{EDB})$
$\mathsf{DB} \leftarrow \mathsf{Apply}(\mathsf{DB}, \mathsf{op}, \mathsf{in})$
$\mathsf{EDB} \leftarrow \mathsf{EDB}', \sigma \leftarrow \sigma'$
if $\sigma' = \texttt{REJECT}$
$\texttt{win} \leftarrow \texttt{true}$

return τ

Figure 5.1: Correctness game for SSE SSECorr.

$\underline{\mathsf{Init}(\mathsf{DB})}$
$(\mathsf{EDB}, K, \sigma) \leftarrow \mathsf{Setup}(\mathsf{DB})$

return EDB

$\underline{\mathsf{Search}(q)}$
$(V, \sigma', \tau) \xleftarrow{\$} \mathsf{Search}_C(K, \sigma, q) \leftrightarrow A$
if $V \neq \texttt{REJECT}$ **then**
$\sigma \leftarrow \sigma'$
if $V \neq \mathsf{DB}(q)$ **then** $\texttt{win} \leftarrow \texttt{true}$
end if

return τ

$\underline{\mathsf{Update}(\mathsf{op}, \mathsf{in})}$
$(\sigma', \tau; \mathsf{EDB}') \xleftarrow{\$}$
$\mathsf{Update}_C(K, \sigma, \mathsf{op}, \mathsf{in}) \leftrightarrow A$
if $\sigma' \neq \texttt{REJECT}$ **then**
$\mathsf{DB} \leftarrow \mathsf{Apply}(\mathsf{DB}, \mathsf{op}, \mathsf{in})$
$\mathsf{EDB} \leftarrow \mathsf{EDB}', \sigma \leftarrow \sigma'$
end if

return τ

Figure 5.2: Soundness game for SSE SSESound.

and SSECorr *is the game defined in Figure 5.1.*

5.4.2 Soundness

To give a proper soundness definition, we use the game SSESound defined in Figure 5.2. The game closely follows the game used to define soundness of interactive provers [Gol04]: the client should not accept an invalid search result. Also, the dynamism of the database raises a difficult point: the verification has to be done over the current version of the database, yet this one must not be modifiable by a malicious server. Hence, SSESound does not apply the update operation on the database when the client rejects the execution of the Update protocol with the server.

Definition 5.2 (SSE Soundness)**.** *An SSE scheme* Π *is* sound *(or* verifiable*) if for all adversary* A*,* $\mathsf{AdvSnd}_A^{\mathsf{SSE},\Pi}(\lambda)$ *is negligible in* λ*, where*

$$\mathsf{AdvSnd}_A^{\mathsf{SSE},\Pi}(\lambda) = \Pr[\mathrm{SSESound}_\Pi^A(\lambda) = 1].$$

5.4.3 Confidentiality

It uses the real world/ideal world paradigm, with games SSEReal and SSEIdeal defined in Figure 5.3, in which the adversary gets from the games, respectively, the real transcripts

of the protocols, or the simulated transcripts, generated by a simulator. The scheme will be secure if no efficient adversary can distinguish between the real and the ideal games.

SSEReal_{Π}

$\underline{\mathsf{Init}(\mathsf{DB})}$
$(\mathsf{EDB}, K, \sigma) \leftarrow \mathsf{Setup}(\mathsf{DB})$

return EDB

$\underline{\mathsf{Search}(q)}$
$(V, \sigma', \tau) \xleftarrow{\$} \mathsf{Search}_C(K, \sigma, q) \leftrightarrow A$
if $V \neq \texttt{REJECT}$
$\sigma \leftarrow \sigma'$

return τ

$\underline{\mathsf{Update}(\mathsf{op}, \mathsf{in})}$
$(\sigma', \tau; \mathsf{EDB}') \xleftarrow{\$} \mathsf{Update}_C(K, \sigma, \mathsf{op}, \mathsf{in}) \leftrightarrow A$
if $\sigma' \neq \texttt{REJECT}$
$\sigma \leftarrow \sigma'$, $\mathsf{EDB} \leftarrow \mathsf{EDB}'$

return τ

$\text{SSEIdeal}_{S,\mathcal{L}}$

$\underline{\mathsf{Init}(\mathsf{DB})}$
$(\mathsf{EDB}, \sigma_S) \leftarrow S(\mathcal{L}^{\mathsf{Stp}}(\mathsf{DB}))$

return EDB

$\underline{\mathsf{Search}(q)}$
$(\sigma'_S, \tau) \xleftarrow{\$} S(\sigma_S, \mathcal{L}^{\mathsf{Srch}}(q)) \leftrightarrow A$
if $\sigma'_S \neq \texttt{REJECT}$
$\sigma_S \leftarrow \sigma'_S$

return τ

$\underline{\mathsf{Update}(\mathsf{op}, \mathsf{in})}$
$(\sigma'_S, \tau) \leftarrow S(\sigma_S, \mathcal{L}^{\mathsf{Updt}}(\mathsf{op}, \mathsf{in})) \leftrightarrow A$
if $\sigma'_S \neq \texttt{REJECT}$
$\sigma_S \leftarrow \sigma'_S$

return τ

Figure 5.3: SSE confidentiality games SSEReal (left) and SSEIdeal (right). The notation $\leftrightarrow A$ represents interactions with the adversary.

Definition 5.3 (SSE Confidentiality)**.** *Let* $\Pi = (\mathsf{Setup}, \mathsf{Search}, \mathsf{Update})$ *be a dynamic SSE instantiation,* A *and* S *probabilistic polynomial-time algorithms, and let* $\mathcal{L}$ *be a stateful algorithm.*

We define the advantage of A *against* S *in the SSE confidentiality security game as* $\mathsf{AdvConf}^{\mathsf{SSE},\Pi}_{A,S,\mathcal{L}}(\lambda)$ *where*

$$\mathsf{AdvConf}^{\mathsf{SSE},\Pi}_{A,S,\mathcal{L}}(\lambda) = \left|\Pr[\text{SSEReal}^{\Pi}_{A}(\lambda) = 1] - \Pr[\text{SSEIdeal}^{\Pi}_{A,S,\mathcal{L}}(\lambda) = 1]\right|.$$

We say that Π *is a* $\mathcal{L}$-adaptively-secure instantiation *if for all adversaries* A*, there exists an efficient algorithm* S *such that* $\mathsf{AdvConf}^{\mathsf{SSE},\Pi}_{A,S,\mathcal{L}}(\lambda) \leq \mathit{negl}(\lambda)$.

5.4.4 E-Set Security Definitions

We give here the formal definition of correctness, confidentiality and soundness of E-Sets, using security games. These games use an auxiliary function Apply that outputs an updated version of X according the input operation op, and element x for that operation.

Definition 5.4. χ *is said to be* correct *if for all adversary* A, $\mathsf{AdvCor}^{\mathsf{ESet},\chi}_{A}(\lambda)$ *is negligible in* λ*, where*

$$\mathsf{AdvCor}^{\mathsf{ESet},\chi}_{A}(\lambda) = \Pr[\text{ESetCorr}^{A}_{\chi}(\lambda) = 1].$$

Definition 5.5 (E-Set confidentiality)**.** *Let* $\chi = (\mathsf{Setup}, \mathsf{Retrieve}, \mathsf{Update})$ *be an E-Set instantiation,* A *and* S *probabilistic polynomial-time algorithms, and let* $\mathcal{L}_E$ *be a stateful algorithm.*

$\underline{\mathsf{Init}(X)}$
$(K_E, \sigma_E, \mathsf{ESet}) \leftarrow \mathsf{Setup}(X)$

return ESet

$\underline{\mathsf{Retrieve}(x)}$
$(b, \sigma_E, \tau) \xleftarrow{\$} \mathsf{Retrieve}(K_E, \sigma_E, x; \mathsf{ESet})$
if $(b = 1 \wedge x \notin X)$ or $(b = 0 \wedge x \in X)$ or $\sigma_E = \texttt{REJECT}$
$\texttt{win} \leftarrow \texttt{true}$

return τ

$\underline{\mathsf{Update}(\mathsf{op}, x)}$
$(\sigma'_E, \tau; \mathsf{ESet}') \xleftarrow{\$} \mathsf{Update}(K_E, \sigma_E, \mathsf{op}, x; \mathsf{ESet})$
$X \leftarrow \mathsf{Apply}(X, \mathsf{op}, x)$
$\mathsf{ESet}' \leftarrow \mathsf{ESet}$, $\sigma'_E \leftarrow \sigma_E$
if $\sigma'_E = \texttt{REJECT}$
$\texttt{win} \leftarrow \texttt{true}$

return τ

Figure 5.4: Correctness game for E-Sets ESetCorr.

We define the advantage of A against S in the E-Set security game as $\mathsf{Adv}^{\mathsf{ESet},\chi}_{A,S,\mathcal{L}_E}(\lambda)$ *where*

$$\mathsf{Adv}^{\mathsf{ESet},\chi}_{A,S,\mathcal{L}_E}(\lambda) = \left|\Pr[\text{ESetReal}^{\chi}_{A}(\lambda) = 1] - \Pr[\text{ESetIdeal}^{\chi}_{A,S,\mathcal{L}_E}(\lambda) = 1]\right|.$$

We say that χ is a $\mathcal{L}_E$-adaptively-secure instantiation *if for all adversaries A, there exists a simulator S such that* $\mathsf{Adv}^{\mathsf{ESet},\chi}_{A,S,\mathcal{L}_E}(\lambda) \leq \mathit{negl}(\lambda)$.

ESetReal$_{\chi}$
$\underline{\mathsf{Init}(X)}$
$(K_E, \sigma, \mathsf{ESet}) \leftarrow \mathsf{Setup}(X)$

return ESet

$\underline{\mathsf{Retrieve}(x)}$
$(b, \sigma', \tau) \xleftarrow{\$} \mathsf{Retrieve}_C(K_E, \sigma, x) \leftrightarrow A$
if $b \neq \texttt{REJECT}$
$\sigma \leftarrow \sigma'$

return τ

$\underline{\mathsf{Update}(\mathsf{op}, x)}$
$(\sigma', \tau; \mathsf{ESet}') \xleftarrow{\$} \mathsf{Update}_C(K_E, \sigma, \mathsf{op}, x) \leftrightarrow A$
if $\sigma' \neq \texttt{REJECT}$
$\sigma \leftarrow \sigma'$, $\mathsf{ESet} \leftarrow \mathsf{ESet}'$

return τ

ESetIdeal$_{S,\mathcal{L}}$
$\underline{\mathsf{Init}(X)}$
$(\mathsf{ESet}, \sigma_S) \leftarrow S(\mathcal{L}^{\mathsf{Stp}}(X))$

return ESet

$\underline{\mathsf{Retrieve}(x)}$
$(\sigma'_S, \tau) \xleftarrow{\$} S(\sigma_S, \mathcal{L}^{\mathsf{Ret}}(x)) \leftrightarrow A$
if $\sigma'_S \neq \texttt{REJECT}$
$\sigma_S \leftarrow \sigma'_S$

return τ

$\underline{\mathsf{Update}(\mathsf{op}, x)}$
$(\sigma'_S, \tau) \xleftarrow{\$} S(\sigma_S, \mathcal{L}^{\mathsf{Updt}}(\mathsf{op}, x)) \leftrightarrow A$
if $\sigma'_S \neq \texttt{REJECT}$
$\sigma_S \leftarrow \sigma'_S$

return τ

Figure 5.5: Security games for E-Sets ESetReal (left) and ESetIdeal (right). The notation $\leftrightarrow A$ represents interactions with the adversary.

Definition 5.6 (E-Set Soundness)**.** *An SSE scheme* Π *is* sound *(or* verifiable*) if for all adversary A,* $\mathsf{AdvSnd}^{\mathsf{ESet},\Pi}_{A}(\lambda)$ *is negligible in λ, where*

$$\mathsf{AdvSnd}^{\mathsf{ESet},\Pi}_{A}(\lambda) = \Pr[\text{ESetSound}^{A}_{\Pi}(\lambda) = 1].$$

$\underline{\mathsf{Init}(X)}$
$(K_E, \sigma, \mathsf{ESet}) \leftarrow \mathsf{Setup}(X)$

return ESet

$\underline{\mathsf{Retrieve}(x)}$
$(b, \sigma', \tau) \xleftarrow{\$}$
$\mathsf{Retrieve}_C(K_E, \sigma, x) \leftrightarrow A$
if $b \neq$ `REJECT` **then**
$\sigma \leftarrow \sigma'$
if $b \neq (x \in X)$ **then** `win` $\leftarrow$ `true`
end if

return τ

$\underline{\mathsf{Update}(\mathsf{op}, x)}$
$(\sigma', \tau; \mathsf{ESet}') \xleftarrow{\$}$
$\mathsf{Update}_C(K_E, \sigma, \mathsf{op}, x) \leftrightarrow A$
if $\sigma' \neq$ `REJECT` **then**
$X \leftarrow \mathsf{Apply}(X, \mathsf{op}, x)$
$\mathsf{ESet} \leftarrow \mathsf{ESet}', \sigma \leftarrow \sigma'$
end if

return τ

Figure 5.6: Soundness game for E-Sets ESetSound.

5.5 A Composite SSE Scheme Supporting Boolean Queries

In this section, we introduce a verifiable multiple-keyword SSE scheme, CXT, for Composite Cross Tag. It builds on top of verifiable SSE and E-Sets instantiations, respectively Π and χ.

The idea behind CXT to support boolean queries is similar to the idea in the OXT protocol in [CJJ+13]: given a query $s \wedge \phi(x_1, \ldots, x_n)$ (under the *Searchable Normal Form* defined in [CJJ+13]), the client will first retrieve the indices of documents containing keyword s and, using their indices, determine which of these also satisfy the query $\phi(x_1, \ldots, x_n)$. The main difference with OXT is that we add an extra interaction: whereas OXT is a one-round protocol, CXT needs (at least) two rounds (depending on the instantiation of single keyword SSE and E-Set we use).

The security of our scheme explicitly relies on the security of the SSE and E-Set primitives, and the leakage function of CXT can be expressed from the leakage functions of Π and χ. Algorithm 14 describes CXT. In this description, we suppose that, every time a sub-protocol returns `REJECT`, the CXT protocol halts and returns `REJECT`.

Security of CXT. Intuitively, CXT leaks exactly what the single-keyword SSE and E-Set instantiations leak. For the Setup and Update procedures, this is immediate. Let us focus on the Search queries. For the SSE part, the leakage is relatively clear: Π encodes DB and is simply queried on keywords. For the E-Set part, the leakage is less obvious, in particular with the properties of E-Set instantiations we emphasized in Section 5.3.2. The set S encodes all the document/keyword pairs. The calls to the retrieve protocol are made on document/keyword pairs where the keywords are $x_1, \ldots, x_n$ and the documents are the documents corresponding to the first keyword s. We know that χ will leak – at most – the repetition of these queries and the result of the queries: the leakage due to χ in a Search query can be expressed as $\mathcal{L}_E(\overline{(x_i, \mathsf{ind})}, (\mathsf{ind} \in \mathsf{DB}(x_i)))$ for all $\mathsf{ind} \in \mathsf{DB}(s)$ and all queried keywords x_i. Recall that the query pattern $\overline{(x_i, \mathsf{ind})}$ takes into account the repeated query elements for both retrieve and update queries.

We can compute the E-Set query pattern incrementally, query by query, as described

Algorithm 14 Description of CXT

Setup(DB)

$S \leftarrow \emptyset$, Parse DB as $(\mathsf{ind}_i, W_i)_{i=1}^d$
for all $w \in W$ **do**
  for all ind $\in \mathsf{DB}(w)$ in rand. order **do**
    $S \leftarrow S \cup \{\mathsf{ind}||w\}$
  end for
end for
$(\mathsf{EDB}_\Pi, K_\Pi, \sigma_\Pi) \leftarrow \Pi.\mathsf{Setup}(\mathsf{DB})$
$(\mathsf{ESet}, K_E, \sigma_E) \leftarrow \chi.\mathsf{Setup}(S)$

return $((\mathsf{EDB}_\Pi, \mathsf{ESet}), (K_\Pi, K_E), (\sigma_\Pi, \sigma_E))$

Search$(K, q, \sigma; \mathsf{EDB})$

Client parses q as $(\phi, s, x_1, \ldots, x_n)$
Client and server run
$V \leftarrow \Pi.\mathsf{Search}(K_\Pi, \sigma_\Pi, s; \mathsf{EDB}_\Pi)$
for all ind $\in V$ in random order, $c = 1, \ldots, |V|$ **do**
  for i = 1 to n **do**
    Client and server run:
    $b[c, i]$
    $\leftarrow \chi.\mathsf{Retrieve}(K_E, \mathsf{ind}||x_i, \sigma_E; \mathsf{ESet})$
  end for
  if $\phi(b[c, 1], \ldots, b[c, n]) = 1$
    output ind
end for

Update$(K, \sigma, \mathsf{op}, \mathsf{in}; \mathsf{EDB})$

Client and server run Π's update protocol:
$(\sigma'_\Pi; \mathsf{EDB}'_\Pi)$
$\leftarrow \Pi.\mathsf{Update}(K_\Pi, \sigma_\Pi, \mathsf{op}, \mathsf{in}; \mathsf{EDB}_\Pi)$
$\mathsf{ESet}' \leftarrow \mathsf{ESet}$, $\sigma'_E \leftarrow \sigma_E$
for all Updated pairs (ind, w) **do**
  Client and server run χ's update protocol:
  $(\sigma'_E; \mathsf{ESet}')$
  $\leftarrow \chi.\mathsf{Update}(K_E, \sigma'_E, \mathsf{op}, \mathsf{ind}||w; \mathsf{ESet}')$
end for

return $((\mathsf{EDB}'_\Pi, \mathsf{ESet}'), (\sigma'_\Pi, \sigma'_E))$

in algorithm 15. The idea is to have a counter that is incremented for every new queried element, and a table that keeps track of the previous queried elements, to compute the query pattern. To the query pattern, we also associate the operation done by the query for an update query, and the result of the membership query for a retrieve query, to form the whole input of the E-Set leakage function XP (for cross pattern). In the interest of simplicity, Algorithm 15 is given the full list of queries issued to the SSE scheme, in a non-adaptive manner, but can easily be re-written adaptively.

The leakage function $\mathcal{L}_{\mathsf{CXT}}$ uses Π's leakage function $\mathcal{L}_\Pi$ and the E-Set leakage function $\mathcal{L}_E$. For the initial query, on input $\mathsf{DB} = (\mathsf{ind}_i, \mathsf{W}_i)_{i=1}^d$, with $|\mathsf{DB}| = N = \sum_{i=1}^d |\mathsf{W}_i|$ $\mathcal{L}_{\mathsf{CXT}}$ returns

$$\mathcal{L}^{\mathsf{Stp}}_{\mathsf{CXT}}(\mathsf{DB}) = \left(\mathcal{L}^{\mathsf{Stp}}_\Pi(\mathsf{DB}), \mathcal{L}^{\mathsf{Stp}}_E(|\mathsf{DB}|)\right).$$

For the i-th query, if it is a Search query, on input $q = (\phi, s, x_1, \ldots, x_n)$ it also computes XP as in algorithm 15. Finally $\mathcal{L}_{\mathsf{CXT}}$ returns

$$\mathcal{L}^{\mathsf{Srch}}_{\mathsf{CXT}}(q) = \left(\mathcal{L}^{\mathsf{Srch}}_\Pi(s), \mathcal{L}^{\mathsf{Ret}}_E(\mathsf{XP}[i])\right)$$

where $\mathcal{L}_E(\mathsf{XP}[i])$ is a shorthand for $\{\mathcal{L}_E(h, b), (h, b) \in \mathsf{XP}[i]\}$.
If it is an Update query, on input $(\mathsf{op}, \mathsf{ind}, w)$, the leakage function also computes XP as defined by algorithm 15 and returns

$$\mathcal{L}^{\mathsf{Updt}}_{\mathsf{CXT}}(\mathsf{op}, \mathsf{ind}, w) = \left(\mathcal{L}^{\mathsf{Updt}}_\Pi(\mathsf{op}, \mathsf{ind}, w), \mathcal{L}^{\mathsf{Updt}}_E(\mathsf{XP}[i])\right)$$

(remember that, when the i-th query is an Update query, $\mathsf{XP}[i]$ is the couple $(\mathsf{op}, \overline{(w, \mathsf{ind})})$).
The formal confidentiality claim for CXT follows. Its full proof is given in 5.5.1.

Algorithm 15 Construction of XP.

Require: DB, a sequence $\mathbf{q}$ of queries.
$H \leftarrow$ empty hash table, $k \leftarrow 0$
for i $= 1$ to $|\mathbf{q}|$ **do**
  $\mathsf{XP}[i] \leftarrow \emptyset$
  if $\mathbf{q}[i]$ is a Search query **then**
    Parse $\mathbf{q}[i]$ as $(s, x_1, \ldots, x_n)$.
    for all $\mathsf{ind} \in \mathsf{DB}(s)$ **do**
      for j **do**1n
        $h_j \leftarrow \textsc{GetIndex}(x_j, \mathsf{ind})$
        $b_j \leftarrow (\mathsf{ind} \in \mathsf{DB}(x_j))$
        $\mathsf{XP}[i] \leftarrow \mathsf{XP}[i] \cup \{(h_j, b_j)\}$
      end for
    end for
  else $\mathbf{q}[i]$ is an Update query
    Parse $\mathbf{q}[i]$ as $(\mathsf{op}, \mathsf{ind}, w)$.
    $h \leftarrow \textsc{GetIndex}(x, \mathsf{ind})$
    $\mathsf{XP}[i] \leftarrow \{(\mathsf{op}, h)\}$
  end if
end for
function $\textsc{GetIndex}(w, \mathsf{ind})$
  if (w, ind) not in H **then**
    $H[w, \mathsf{ind}] \leftarrow k$
    $k \leftarrow k + 1$
  end if

  return $H[w, \mathsf{ind}]$
end function

Theorem 5.7. *If* Π *is a* $\mathcal{L}_\Pi$*-adaptively-secure SSE scheme and* χ *is a content-hiding* $\mathcal{L}_E$*-adaptively-secure E-Set instantiation then* CXT *is* $\mathcal{L}_{\mathsf{CXT}}$*-adaptively-secure.*

5.5.1 Proof of CXT Confidentiality (Theorem 5.7)

We recall the leakage function $\mathcal{L}_{\mathsf{CXT}}$: for the initial call, on input $\mathsf{DB} = (\mathsf{ind}_i, \mathsf{W}_i)_{i=1}^d$, $\mathcal{L}_{\mathsf{CXT}}$ computes the array $\mathbf{T}$ indexed over the keywords W, such that $\mathbf{T}[w]$ is a list containing the elements in $\mathsf{DB}(w)$, randomly permuted, $N = \sum_{i=1}^d |\mathsf{W}_i|$, and returns

$$\mathcal{L}^{\mathsf{Stp}}_{\mathsf{CXT}}(\mathsf{DB}) = (\mathcal{L}^{\mathsf{Stp}}_{\Pi}(\mathsf{DB}), \mathcal{L}^{\mathsf{Stp}}_{E}(N)).$$

If the i-th query is a Search query $(\phi, s, x_1, \ldots, x_n)$, $\mathcal{L}_{\mathsf{CXT}}$ computes the repeat pattern for all the couples (ind, x_i) for $\mathsf{ind} \in \mathsf{DB}(s)$, using the algorithm 15. $\mathcal{L}_{\mathsf{CXT}}$ the outputs

$$\mathcal{L}^{\mathsf{Srch}}_{\mathsf{CXT}}(\phi, s, x_1, \ldots, x_n) = (\mathcal{L}^{\mathsf{Srch}}_{\Pi}(s), \{\mathcal{L}^{\mathsf{Ret}}_{E}(h, b), (h, b) \in \mathsf{XP}[i]\}).$$

If the i-th query is an Update query $(\mathsf{op}, \mathsf{ind}, w)$, once again the $\mathcal{L}_{\mathsf{CXT}}$ uses algorithm 15 to compute the query pattern of (ind, w). $\mathcal{L}_{\mathsf{CXT}}$ the outputs

$$\mathcal{L}^{\mathsf{Updt}}_{\mathsf{CXT}}(\mathsf{op}, \mathsf{ind}, w) = \left(\mathcal{L}^{\mathsf{Updt}}_{\Pi}(\mathsf{op}, \mathsf{ind}, w), \mathcal{L}^{\mathsf{Updt}}_{E}(\mathsf{XP}[i])\right).$$

Theorem 5.8. CXT *is* $\mathcal{L}_{\mathsf{CXT}}$*-adaptively-secure against adaptive adversary.*

Proof. We adopt a similar strategy to [CJJ+13] to show the security: we will use successive games, and show that they are indistinguishable.

Game G_0 will generate exactly the same transcript distribution as $\text{SSEReal}^{\mathsf{CXT}}_A(\lambda)$. The last game will generate an indistinguishable transcript only given the output of the leakage function $\mathcal{L}_{\mathsf{CXT}}$. Game G_0 is actually the SSE real security game unrolled with our CXT instantiation, as described in figure 5.7.

Game G_0. Game G_0 does exactly the same thing as the real game, plus some extra record-keeping: the hash table H records the repetitions of the couples (ind, x_i).

In the game description, we omit to mention the Final function as it does not change: it always returns the bit passed in argument. We have

$$\Pr[G_0 = 1] = \Pr[\text{SSEReal}^{\mathsf{CXT}}_A(\lambda) = 1]$$

$\underline{\mathsf{Init}(\mathsf{DB})}$

Parse DB as $(\mathsf{ind}_i, W_i)_{i=1}^d$
$S \leftarrow \emptyset$
for all $w \in W$ **do**
  for all $\mathsf{ind} \in \mathsf{DB}(w)$ in rand. order **do**
    $S \leftarrow S \cup \{\mathsf{ind}||w\}$
  end for
end for
$(\mathsf{EDB}_\Pi, K_\Pi, \sigma_\Pi) \leftarrow \Pi.\mathsf{Setup}(\mathsf{DB})$
$(\mathsf{ESet}, K_E, \sigma_E) \leftarrow \chi.\mathsf{Setup}(S)$
$H \leftarrow$ empty hash table, $k \leftarrow 0$

return $(\mathsf{EDB}_\Pi, \mathsf{ESet})$

$\underline{\mathsf{Update}(\mathsf{op}, w, \mathsf{ind})}$

$(\sigma'_\Pi, \tau_\Pi; \mathsf{EDB}'_\Pi) \xleftarrow{\$}$
  $\Pi.\mathsf{Update}_C(K_\Pi, \sigma_\Pi, \mathsf{op}, \mathsf{in}) \leftrightarrow A$
if $\sigma'_\Pi = \texttt{REJECT}$

return τ_Π
$i \leftarrow 0$, $\sigma'_E \leftarrow \sigma_E$
for all Updated pairs (ind, w); i++ **do**
  if (ind, w) not in H **then**
    $H[\mathsf{ind}, w] \leftarrow k$
    $k \leftarrow k + 1$
  end if
  $(\sigma'_E, \tau_E^i; \mathsf{ESet}) \xleftarrow{\$}$
    $\chi.\mathsf{Update}_C(K_E, \sigma'_E, \mathsf{op}, \mathsf{ind}||w) \leftrightarrow A$
  if $\sigma'_E = \texttt{REJECT}$

  return $(\tau_\Pi, \{\tau_{\mathsf{ESet}}^j\}_{j=1}^i)$
end for
$\sigma_\Pi \leftarrow \sigma'_\Pi, \sigma_E \leftarrow \sigma'_E$

return $(\tau_\Pi, \{\tau_{\mathsf{ESet}}^j\}_{j=1}^i)$

$\underline{\mathsf{Search}(\phi, s, x_1, \ldots, x_n)}$

$(V, \sigma'_\Pi, \tau_\Pi) \xleftarrow{\$} \Pi.\mathsf{Search}_C(K_\Pi, \sigma_\Pi, w) \leftrightarrow A$
$\boxed{\{\mathsf{ind}_1, \ldots, \mathsf{ind}_m\} \leftarrow \mathsf{DB}(s)}$
if $V = \texttt{REJECT}$

return τ_Π
for c **do**1|V|
  $\sigma'_E \leftarrow \sigma_E$, $\mathcal{T} \leftarrow \emptyset$
  Pick new random $\mathsf{ind} \xleftarrow{\$} V$
  $\boxed{\text{Pick new random } \mathsf{ind} \xleftarrow{\$} \mathsf{DB}(s)}$
  for i **do**1n
    if (ind, x_i) not in H **then**
      $H[\mathsf{ind}, x_i] \leftarrow k$
      $k \leftarrow k + 1$
    end if
    $(b[c, i], \sigma'_E, \tau_E) \xleftarrow{\$}$
      $\chi.\mathsf{Retrieve}_C(K_E, \sigma'_E, \mathsf{ind}||x_i) \leftrightarrow A$
    $\mathcal{T} \leftarrow \mathcal{T} \cup \{\tau_E\}$
    if $\sigma'_E = \texttt{REJECT}$

    return $(\tau_\Pi, \mathcal{T})$
  end for
end for
$\sigma_\Pi \leftarrow \sigma'_\Pi$, $\sigma_E \leftarrow \sigma'_E$

return $(\tau_\Pi, \mathcal{T})$

Figure 5.7: Games G_0 and G_1. Boxed code is included in G_1 only.

Game G_1. In game G_1, we no longer get the indices ind_i from the set V but directly from DB. Let m be the size of $\mathsf{DB}(s)$. Values of $\mathsf{DB}(s)$ are used instead of V only if the client did not rejected the result of Π's search query. Hence, the advantage of distinguishing between games G_1 and G_0 is the advantage in succeeding in SSE soundness game (*cf.* section 5.2.4). We can show that there is a ppt adversary B_1 such that

$$\Pr[G_1 = 1] - \Pr[G_0 = 1] \leq \mathsf{AdvSnd}_{B_1}^{\mathsf{SSE},\Pi}(\lambda).$$

Game G_2. In game G_2, we replace the calls to Π instantiation by calls to the simulator S_Π. S_Π is chosen given the following SSE adversary B_2: in the SSE confidentiality games, B_2 starts by generating EDB_Π as in game G_1. Then, it forwards game G_1 queries:

$\underline{\mathsf{Init}(\mathsf{DB})}$

Parse DB as $(\mathsf{ind}_i, W_i)_{i=1}^d$
$S \leftarrow \emptyset$
for all $w \in W$ **do**
  for all $\mathsf{ind} \in \mathsf{DB}(w)$ in rand. order **do**
    $S \leftarrow S \cup \{\mathsf{ind}||w\}$
  end for
end for
$(\mathsf{EDB}_\Pi, \sigma_\Pi) \leftarrow S_\Pi(\mathcal{L}_\Pi^{\mathsf{Updt}}(\mathsf{DB}))$
$(\mathsf{ESet}, K_E, \sigma_E) \leftarrow \chi.\mathsf{Setup}(S)$
$\boxed{(\mathsf{ESet}, \sigma_E) \leftarrow S_E(\mathcal{L}_E^{\mathsf{Updt}}(\mathsf{DB}))}$
$H \leftarrow$ empty hash table, $k \leftarrow 0$

return $(\mathsf{EDB}_\Pi, \mathsf{ESet})$

$\underline{\mathsf{Update}(\mathsf{op}, w, \mathsf{ind})}$

$(\sigma_\Pi, \tau_\Pi) \xleftarrow{\$} S_\Pi(\sigma_\Pi, \mathcal{L}_\Pi^{\mathsf{Updt}}(\mathsf{op}, \mathsf{in}, \mathsf{ind})) \leftrightarrow A$
if S_Π did not `REJECT`

return τ_Π
$i \leftarrow 0$, $\sigma'_E \leftarrow \sigma_E$
for all Updated pairs (ind, w); i++ **do**
  if (ind, w) not in H **then**
    $H[\mathsf{ind}, w] \leftarrow k$
    $k \leftarrow k + 1$
  end if
  $(\sigma'_E, \tau_E^i; \mathsf{ESet}) \xleftarrow{\$}$
    $\chi.\mathsf{Update}_C(K_E, \sigma'_E, \mathsf{op}, \mathsf{ind}||w) \leftrightarrow A$
  $\boxed{(\sigma'_E, \tau_E^i) \xleftarrow{\$} S_E(\sigma'_E, \mathcal{L}_E^{\mathsf{Updt}}(\mathsf{op}, H[\mathsf{ind}, w])) \leftrightarrow A}$
  if $\sigma'_E =$ `REJECT`

  return $(\tau_\Pi, \{\tau_{\mathsf{ESet}}^j\}_{j=1}^i)$
end for
$\sigma_\Pi \leftarrow \sigma'_\Pi, \sigma_E \leftarrow \sigma'_E$

return $(\tau_\Pi, \{\tau_{\mathsf{ESet}}^j\}_{j=1}^i)$

$\underline{\mathsf{Search}(\phi, s, x_1, \ldots, x_n)}$

$(\sigma'_\Pi, \tau_\Pi) \leftarrow S_\Pi(\sigma_\Pi, \mathcal{L}_\Pi^{\mathsf{Srch}}(w))$
$\{\mathsf{ind}_1, \ldots, \mathsf{ind}_m\} \leftarrow \mathsf{DB}(s)$
if S_Π did not `REJECT`

return τ_Π
$\sigma'_E \leftarrow \sigma_E$, $\mathcal{T} \leftarrow \emptyset$
for c **do**1|V|
  Pick new random $\mathsf{ind} \xleftarrow{\$} \mathsf{DB}(s)$
  for i **do**1n
    if (ind, x_i) not in H **then**
      $H[\mathsf{ind}, x_i] \leftarrow k$
      $k \leftarrow k + 1$
    end if
    $(b[c,i], \sigma'_E, \tau_E) \xleftarrow{\$}$
      $\chi.\mathsf{Retrieve}_C(K_E, \sigma_E, \mathsf{ind}||x_i) \leftrightarrow A$
    $\boxed{b[c,i] \leftarrow 1 \text{ if } \mathsf{ind} \in \mathsf{DB}(x_i), 0 \text{ otherwise}}$
    $\boxed{(\sigma'_E, \tau_E) \leftarrow S_E(\sigma'_E, \mathcal{L}_E^{\mathsf{Ret}}(H[\mathsf{ind}, x_i], b[c,i]))}$
    $\mathcal{T} \leftarrow \mathcal{T} \cup \{\tau_E\}$
  end for
end for
$\sigma_\Pi \leftarrow \sigma'_\Pi$, $\sigma_E \leftarrow \sigma'_E$

return $(\tau_\Pi, \mathcal{T})$

Figure 5.8: Games G_2 and G_3. Boxed code is included in G_3 only.

when $\mathsf{Search}(\phi, s, x_1, \ldots, x_n)$ is queried in G_1, B_2 queries $\mathsf{Search}(s)$, when G_1 queries $\mathsf{Update}(\mathsf{op}, w, \mathsf{ind})$, B_2 queries $\mathsf{Update}(\mathsf{op}, w, \mathsf{ind})$. Finally B_2 outputs A's output. As Π is $\mathcal{L}_\Pi$ adaptively secure against adaptive adversaries, there exists an efficient simulator S_Π for B_2 in the ideal SSE security game.

Hence,

$$\Pr[G_2 = 1] - \Pr[G_1 = 1] \leq \mathsf{Adv}_{B_3, S_\Pi, \mathcal{L}_\Pi}^{\mathsf{SSE}, \Pi}(\lambda).$$

Game G_3. In game G_3, we replaced the calls to the E-Set instantiation χ by calls to the simulator S_E. S_E is constructed as in the indistinguishability proof for G_2.

The adversary B_3 for the E-Set games starts by generating X as in game G_2, and forwards queries as previously: when $\mathsf{Search}(\phi, s, x_1, \ldots, x_n)$ is queried in G_2, B_3 queries $\mathsf{Retrieve}(\mathsf{ind}||x_i)$ for $\mathsf{ind} \in \mathsf{DB}(s)$ and $i = 1, \ldots, n$. When G_2 queries $\mathsf{Update}(\mathsf{op}, w, \mathsf{ind})$, B_3 queries $\mathsf{Update}(\mathsf{op}, \mathsf{ind}||w)$. Finally B_3 outputs A's output. As χ is $\mathcal{L}_E$ adaptively secure against adaptive adversaries, there exists an efficient simulator S_E for B_3 in the ideal E-Set security game.

$$\Pr[G_3 = 1] - \Pr[G_2 = 1] \leq \mathsf{Adv}^{\mathsf{ESet},\chi}_{B_3,S_E,\mathcal{L}_E}(\lambda).$$

The ideal game and conclusion. All the outputs of game G_3 are generated by simulators S_Π or S_E given some leakage function's input. The input to S_Π is $\mathcal{L}_\Pi$ which is explicitly given by $\mathcal{L}_{\mathsf{CXT}}$. The input to S_E is $\mathcal{L}_E$. In game G_4, what is given to $\mathcal{L}_E$ is actually the same as the $\mathcal{L}_E$ part of the $\mathcal{L}_{\mathsf{CXT}}$ function: XP is computed the same way as $(\mathsf{op}, H[\mathsf{ind}, w])$ or $(H[\mathsf{ind}, x_i], b[c, i])$.

Finally, the simulator S just applies S_T to the first element of the leakage function $\mathcal{L}_{\mathsf{CXT}}$ and then S_T to the other elements, which is exactly what G_3 does.

Hence,

$$\Pr[G_3 = 1] = \Pr[\text{SSEIdeal}^{\mathsf{CXT}}_{A,S,\mathcal{L}_{\mathsf{CXT}}}(\lambda) = 1]$$

Once we combine the elements of the proofs, we obtain that

$$\mathsf{Adv}^{\mathrm{SSE},\mathsf{CXT}}_{A,S,\mathcal{L}_{\mathsf{CXT}}}(\lambda) \leq \mathsf{AdvSnd}^{\mathsf{SSE},\Pi}_{B_1}(\lambda) + \mathsf{Adv}^{\mathsf{SSE},\Pi}_{A,B_2,\mathcal{L}_\Pi}(\lambda) + \mathsf{Adv}^{\mathsf{ESet},\chi}_{A,B_3,\mathcal{L}_E}(\lambda).$$

CXT is $\mathcal{L}_{\mathsf{CXT}}$-adaptively-secure if Π is a $\mathcal{L}_\Pi$-adaptively-secure against adaptive adversary single keyword SSE instantiation and χ is a content-hiding $\mathcal{L}_E$-adaptively-secure against adaptive adversary E-Set instantiation.

□

Similarly, the soundness of CXT is inherited from Π's and χ's soundness. In 5.5.2, we show the following theorem:

Theorem 5.9 (Soundness of CXT)**.** *If Π is a sound SSE instantiation, and χ a sound E-Set instantiation, then CXT is a sound SSE scheme supporting boolean queries.*

5.5.2 Proof of CXT Soundness (Theorem 5.9)

Proof. To prove soundness of CXT, we use the following hybrids:

1. H_0 is the original SSESound game.
2. H_1 is H_0 where Π is replaced by an ideal single keyword SSE scheme $\widetilde{\Pi}$ such that $\widetilde{\Pi}.\mathsf{Search}(K_\Pi, \sigma_\Pi, s; \mathsf{EDB}_\Pi) = \mathsf{DB}(s)$ or `REJECT` (when the server is cheating).
3. H_2 is H_1 where χ is replaced by an ideal E-Set $\widetilde{\chi}$ such that $\widetilde{\chi}.\mathsf{Retrieve}(K_E, \sigma_E, \mathsf{ind}||x_i; \mathsf{ESet}) = (\mathsf{ind} \in \mathsf{DB}(x_i))$ or `REJECT` (for a cheating server).

   By construction, H_2 always returns the correct results or `REJECT`. Hence, it is totally sound: for every adversary A, $\Pr[H_2^A(\lambda) = 1] = 0$. Finally, we have that, for every

adversary A, there exist two adversaries B and C such that

$$\begin{aligned}\mathsf{AdvSnd}_A^{\mathsf{SSE},\mathsf{CXT}}(\lambda) &= \Pr[H_0^A(\lambda) = 1] \\ &= \Pr[H_0^A(\lambda) = 1] - \Pr[H_1^A(\lambda) = 1] \\ &\quad + \Pr[H_1^A(\lambda) = 1] - \Pr[H_2^A(\lambda) = 1] \\ &\quad + \Pr[H_2^A(\lambda) = 1] \\ &= \mathsf{AdvSnd}_B^{\mathsf{SSE},\Pi}(\lambda) + \mathsf{AdvSnd}_C^{\mathsf{ESet},\chi}(\lambda)\end{aligned}$$

□

Performances. We can notice that the calls to the E-Set protocols can be done concurrently. Hence, the scheme essentially performs $I_\Pi + I_\chi$ interactions per search query, where I_Π (resp. I_χ) is the number of interactions needed by Π (resp. χ) to run the Search (resp. Retrieve) protocol.
Concerning computation complexity, the server performs $C_\Pi + n \cdot |\mathsf{DB}(s)| \cdot C_\chi$ operations for a search query, where C_Π (resp. C_χ) is the number of operations needed by Π (resp. χ) to run the Search (resp. Retrieve) protocol, and n the number of keywords in the conjunctive formula ϕ. Hence, if both Π and χ are sublinear, CXT is sublinear.

Security/Performance Tradeoff. We can reduce the amount of leakage of CXT at the cost of increased space. In the original CXT scheme, the E-Set is filled with keyword/document pairs and queried on pairs of conjunctive search keywords (the x_i) and documents matching the first keyword.

In the modified scheme, CXXT (for Composite Cross Cross Tag), the E-Set encodes the set of triples $S = \{(w_1, w_2, \mathsf{ind}) | \mathsf{ind} \in \mathsf{DB}(w_1) \cap \mathsf{DB}(w_2)\}$, and queried on (s, x_i, ind) for $\mathsf{ind} \in \mathsf{DB}(s)$. Now, query repetitions will happen only when (s, x_i, ind) repeats, *i.e.* only when (s, x_i) repeats. Hence, for the E-Set part, CXXT will only leak the query pattern of the whole search query. However, this has a cost: the E-Set now has to store $\sum_{i=1}^{d} |\mathsf{W}_i|^2$ entries instead of $\sum_{i=1}^{d} |\mathsf{W}_i|$, which will not only increase space complexity, but might also increase the query complexity.

5.6 CXT Instantiations

In this section, we describe how to instantiate the single-keyword SSE and E-Set primitives to construct a verifiable dynamic SSE scheme supporting verifiable conjunctive boolean queries.

5.6.1 Verifiable Single-Keyword SSE

To instantiate Π in the CXT construction, we can rely on the work of Kurosawa and Ohtaki [KO13] or of Bost *et al.* [BFP16] (*cf.* Section 5.3.1).

Generic SSE Verification (GSV - [BFP16]). We quickly recall that GSV is a modular construction, that bases itself on three key primitives: a (non-verifiable) dynamic single-keyword SSE scheme Π, a verifiable hash table (VHT), a set hashing scheme.

Table 5.1: Leakage for different single keyword verifiable SSE schemes. $\mathcal{L}_\Pi$ is the leakage function of GSV's underlying SSE scheme Π. The second line gives the leakage when Π is $\Pi_{\mathsf{bas}}^{\mathsf{dyn}}$ from [CJJ+14]. In this case, the leakage pattern has been amplified for the sake of simplicity. For [GMP15]'s update leakage, μ is the number of modified keywords.

	$\mathcal{L}^{\mathsf{Stp}}(\mathsf{DB})$	$\mathcal{L}^{\mathsf{Srch}}(\mathsf{DB}, w)$	$\mathcal{L}^{\mathsf{Updt}}(\mathsf{DB}, \mathsf{op}, \mathsf{in})$
GSV [BFP16]	$(\mathcal{L}_\Pi^{\mathsf{Stp}}(\mathsf{DB}), \lvert W\rvert)$	$(\mathcal{L}_\Pi^{\mathsf{Srch}}(\mathsf{DB}, q), \overline{w})$	$(\mathcal{L}_\Pi^{\mathsf{Updt}}(\mathsf{DB}, \mathsf{op}, \mathsf{in}), \mathsf{op}, \overline{w})$
GSV $-\ \Pi_{\mathsf{bas}}^{\mathsf{dyn}}$	$(N, \lvert W\rvert)$	$(\mathsf{HistDB}(w), \overline{w})$	$(\mathsf{op}, \mathsf{HistDB}(w), \overline{w})$
[KO13]	$(N, \lvert W\rvert)$	$(\mathsf{HistDB}(w), \overline{w})$	$(\mathsf{op}, \mathsf{HistDB}(w), \overline{w})$
[GMP15]	$(N, \lvert W\rvert)$	$\lvert\mathsf{DB}(w)\rvert$	μ

The first one is easy to get from previous works [KPR12, CJJ+14]. A verifiable hash table is similar to a regular hash table, but also has a soundness property: when querying a key mapping to an element, the server proves the returned element is the correct one, and that there are no associated element when querying a key that is not present in the hash table. Verifiable hash tables have extensively been studied in the past (*e.g.* [PTT09, TT05]). In Section 5.6.3, we give a simple construction based on binary search trees and hash trees that we used for our implementation, similar in spirit to the one in [TT05].

A set hash function is a hash function that takes sets as input, and that can be computed incrementally: from the hash of S and S' (with $S \cap S' = \emptyset$), we can easily compute the hash of $S \cup S'$. ECMH [MSTA16] is such a set hash functions.

The idea of GSV is to hash the sets $\mathsf{DB}(w)$ for all keyword $w \in W$ and store the hash value in a VHT. To verify the results of a search query, the client will compute the hash of the set of responses he got from the server, and check that it matches the value stored in the VHT. Updates can be efficiently performed using the homomorphic properties of the set hash function.

Hiding Document Indices. Most of the previous constructions cannot be used as is, as the server learns the result indices: when keyword w is searched, the leakage function returns (among other things) $\mathsf{DB}(w)$.

In our construction, this is unacceptable: revealing the indices corresponding to the s term of a search query, not only the results of the full query, is not acceptable. These intermediate results have to be hidden. We propose to use deterministic encryption, *i.e.* use a PRP to hide the indices, with a keyword-dependent key. This will prevent the direct leakage of the indices, but will leak the repetition of document/keyword pairs. In the case of GSV using the dynamic construction in [CJJ+14] (as the unverified SSE scheme), this is not a problem as the leakage function already leaks both the repetition of searched (resp. updated) keywords, and the indices of searched (resp. updated) documents. In particular, it does not have this *forward security* property that prevents an inserted/removed document index to leak if a search over a keyword it contains was executed before.

If we still want to achieve forward security, we can also use the ORAM-based SSE scheme of Garg *et al.* [GMP15]. Indeed, this scheme leaks only size information (number of pairs, keywords, and results). Unfortunately, as is, the scheme is not secure against malicious adversaries. This issue can easily be overcome by using Path ORAM verification techniques [RFY+13]. Note that [GMP15] only supports additions, not deletions.

The leakage functions for these constructions are given in Table 5.1.

5.6.2 E-Set Instantiation

It is easy to construct verifiable sets from verifiable hash tables, by using void mapped values. The E-Set's soundness will be directly inherited from the VHT's soundness. And as explained in Section 5.3.3, we can easily construct an E-Set scheme whose leakage function is

$$\mathcal{L}_E^{\mathsf{Stp}}(X) = |X|, \mathcal{L}_E^{\mathsf{Ret}}(x) = (\overline{x}, x \in X), \text{ and } \mathcal{L}_E^{\mathsf{Updt}}(\mathsf{op}, x) = (\mathsf{op}, \overline{x}).$$

However, this is not suitable if we want our scheme to be forward secure: the keys the protocol will query (either for searches or updates) are deterministically generated from the index/keyword pair, and a membership query for a document/keyword pair followed by a modification of this pair would immediately be spotted by the server, breaking its forward security. A solution to this problem is to use oblivious data structures as constructed in [WNL+14], which, as they are inspired from Path-ORAM, can be easily authenticated (*cf.* [RFY+13]). This way, we end up with a forward-secure verifiable set, and associated with a forward-secure single-keyword SSE scheme, we get that CXT itself is forward-secure.

5.6.3 Verifiable Hash Table Instantiation

We give here a very simple and memory efficient instantiation of a verifiable hash table. It is heavily inspired in [CHKO08] from existing hash tree constructions, associated with binary search trees. Here, we aim for simplicity and memory efficiency.

We use a (collision-resistant) hash function $h : \{0,1\}^* \to \{0,1\}^\lambda$. Each node n of a tree T is a tuple $(\mathsf{hkey}, v, c_l, c_r)$ where $\mathsf{hkey} \in \{0,1\}^\lambda$ is the *key* of the node, v its *value*, and c_l and c_r are the respectively left and right children of n, $c_l, c_n \in T \cup \{\bot\}$. For each node, we recursively define its hash as $H(n) := h(\mathsf{hkey}||v||H(c_l)||H(c_r))$, and $H(\bot) := h(0)$. Let $root(T)$ be the root of T, and for a node $n \in T$, $T(n)$ is the subtree of T rooted at n.

We want to maintain the invariant that T is a binary search tree: for every $n \in T$, keys of the subtree rooted at $n.c_l$ are smaller than $n.\mathsf{hkey}$, and keys of the subtree rooted at $n.c_r$ are larger than $n.\mathsf{hkey}$.

We also define an algorithm Find that, given a tree T and input x, finds the couple of nodes $(n_{\texttt{big}}, n_{\texttt{small}}) \in (T \cup \{\bot\})^2$ such that

- $(n_{\texttt{big}}, n_{\texttt{small}}) \neq (\bot, \bot)$;
- if $n_{\texttt{big}} \neq \bot$ then $n_{\texttt{big}}.\mathsf{hkey} \leq x$ and $n_{\texttt{big}}.c_l.\mathsf{hkey} < x$ or $n_{\texttt{big}}.c_l = \bot$;
- if $n_{\texttt{small}} \neq \bot$ then $n_{\texttt{small}}.\mathsf{hkey} \geq x$ and $n_{\texttt{small}}.c_r.\mathsf{hkey} > x$ or $n_{\texttt{small}}.c_r = \bot$;
- $n_{\texttt{big}} = n_{\texttt{small}}$ iff $x = n_{\texttt{big}}.\mathsf{hkey}$ (a node with key x is in T).

We call this pair of nodes, the *enclosing nodes*. Find also builds a list L of tuples in $(T \cup \{\bot\})^2 \times \{\mathsf{l}, \mathsf{r}\}$ which represents the path followed from the root to the enclosing nodes. The pseudo-code of Find is given in Algorithm 16, and Figure 5.9 gives two examples of application of this algorithm.

Together with the root of the tree T, the list L can be used as a proof, to show that $n_{\texttt{big}}$ and $n_{\texttt{small}}$ are actually the enclosing nodes of key x, by proceeding as we would do with a Merkle hash tree (recomputing the hash of the root given the hash of the elements of the list). This is exactly what a server has to prove to a client asking for key x. But this is also very useful for update queries: when inserting a new key-value pair (hkey, v),

Algorithm 16 Find algorithm

```
1: Find(T, x)
2: larger_node ← ⊥
3: smaller_node ← ⊥
4: current ← root(T)
5: L ← ε
6: while current ≠ ⊥ do
7:     if current.hkey = x then
                                   ▷ Found x. Halt.
8:         larger_node ← current
9:         smaller_node ← current
10:        break
11:    else if current.hkey > x then
           ▷ We will go left, add the current node and its
       right node to L.
12:        L.enqueue(current, current.c_r, l)
13:        larger_node ← current
14:        current ← current.c_l
15:    else if current.hkey < x then
           ▷ We will go right, add the current node and its
       left node to L.
16:        L.enqueue(current, current.c_l, r)
17:        smaller_node ← current
18:        current ← current.c_r
19:    end if
20: end while
21:
22: return (larger_node, smaller_node, L)
```

the client will be able to ensure that the node as been correctly inserted in the tree (*i.e.* that the binary search tree invariant is kept), and to update its copy of the hash value of the root. Similarly, when updating pair (hkey, v) to (hkey, v') the server will show that the right node as been selected, and the client will be able to update the hash value of the root. Finally, deletion is a bit more involved. Suppose we want to delete node n with key hkey. Two cases have to be considered, depending on the number of children of n:

- n has zero or one child. The deletion is also very easy: we remove the node, replace it with its child (if it exists) and update the hashes of its ancestors. The client can easily recompute the hash of the root from the list L generated by $\mathsf{Find}(T, \mathsf{hkey})$.

- n has two children. Find the in-order successor (or predecessor) n_+ of n, copy n_+'s key/value pair of to n and delete n_+. Deleting n_+ will fall in one of the first two cases (if n_+ has two children, the left child's value will be less than the one of n_+ yet be bigger than hkey by construction of the binary search tree). It is easy to find n_+ using Find: running $\mathsf{Find}(T, \mathsf{hkey} + 1)$ will return the tuple (n_+, n, L), from which it will be easy to recompute the hash of the root.

5.6.4 Security and Performance

We can apply Theorem 5.7 to the previous dynamic instantiations to build a dynamic SSE scheme supporting boolean queries.

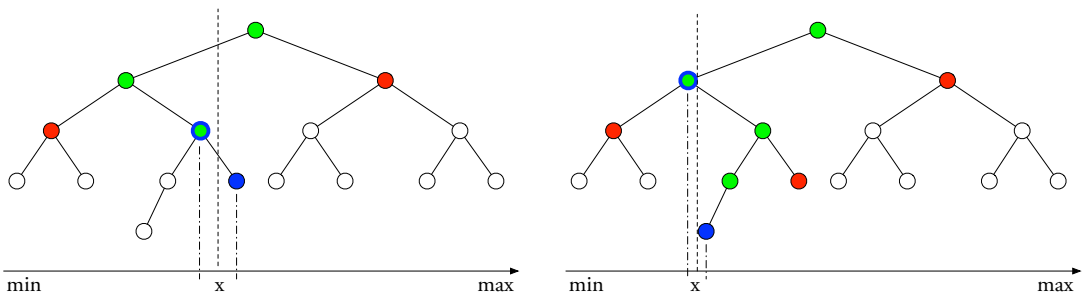


Figure 5.9: Representation of an execution of Find. Blue nodes represent the nodes returned by the search queries, *i.e.* the results of the search. Green and red nodes are part of the proof as elements of the list L. For a tuple $(n, n_c, .)$ in L, n is in green and n_c in red.

Table 5.2: Complexity of multiple keywords schemes. The O notation hides the polynomial dependency in the security parameter, $\widetilde{O}$ the log log factors. μ is the number of updated document/keyword pairs. Rounds are given for searches / updates. [CJJ+13] is given as a comparison reference. As is, it does not support updates.

	Search	Update	Rounds	Space
[CJJ+13]	$O(m)$	-	1 / -	$O(N)$
$\mathsf{CXT}_{\mathsf{bas}}$	$O(m \log N + \log \lvert W \rvert)$	$O(\mu(\log \lvert W \rvert + \log N))$	2 / 1	$O(N)$
$\mathsf{CXT}_{\mathsf{min}}$	$\widetilde{O}(m \log N + \log^3 \lvert W \rvert)$	$\widetilde{O}(\mu(\log^3 \lvert W \rvert + \log N))$	4μ / 2μ	$O(N)$

Corollary 5.10. *When instantiated with* GSV *and hash trees (resp. sections 5.6.1 and 5.6.2),* $\mathsf{CXT}_{\mathsf{bas}}$ *is* $\mathcal{L}^{\mathsf{bool}}_{\mathsf{bas}}$*-adaptively-secure against adaptive adversary, with*

$$\begin{aligned}
\mathcal{L}^{\mathsf{Stp}}_{\mathsf{bas}}(\mathsf{DB}) &= (N, |W|) \text{ for the initialization leakage} \\
\mathcal{L}^{\mathsf{Srch}}_{\mathsf{bas}}(\phi, s, x_1, \ldots, x_n) &= (\overline{s}, |\mathsf{DB}(s)|, \mathsf{XP}[i]) \text{ for the search leakage of the } i\text{-th query} \\
&= \Big(\overline{s}, |\mathsf{DB}(s)|, \Big\{\overline{(s, \mathsf{ind})}, \mathsf{ind} \in \mathsf{HistDB}(s)\Big\}, \\
&\qquad \Big\{(\mathsf{ind} \in \mathsf{DB}(x_i)), \overline{(x_i, \mathsf{ind})}), \mathsf{ind} \in \mathsf{DB}(s)\Big\}_{i=1}^{n}\Big) \\
\mathcal{L}^{\mathsf{Updt}}_{\mathsf{bas}}(\mathsf{op}, W^{\mathsf{Updt}}, \mathsf{ind}) &= \Big(\mathsf{op}, \Big\{\Big(\overline{w}, \overline{(w, \mathsf{ind})}\Big), w \in W^{\mathsf{Updt}}\Big\}\Big)
\end{aligned}$$

Corollary 5.11. *When instantiated with oblivious data structures (such as [GMP15] for the SKS scheme and [WNL+14] for the E-Set, and using [RFY+13] to ensure soundness),* $\mathsf{CXT}_{\mathsf{min}}$ *(without support for deletions) is* $\mathcal{L}_{\mathsf{min}}$*-adaptively-secure against adaptive adversary, with*

$$\begin{aligned}
\mathcal{L}^{\mathsf{Stp}}_{\mathsf{min}}(\mathsf{DB}) &= (N, |W|), \\
\mathcal{L}^{\mathsf{Srch}}_{\mathsf{min}}(\phi, s, x_1, \ldots, x_n) &= (|\mathsf{DB}(s)|, n), \text{ and} \\
\mathcal{L}^{\mathsf{Updt}}_{\mathsf{min}}(\mathsf{add}, W^+, \mathsf{ind}) &= (|W^+|).
\end{aligned}$$

Complexity. As we saw earlier (Section 5.5.2), the performances of this new scheme can immediately be inferred from the performances of the single-keyword SSE and the E-Set. Table 5.2 gives the complexity for the previous two instantiations of CXT.

5.7 Implementation and Evaluation

Implementation. We have prototyped, in C/C++, the CXT scheme using the basic instantiation of Section 5.6, named $\mathsf{CXT}_{\mathsf{bas}}$. The Open Source implementation is available at [BC16]. The cryptographic primitives we used are Blake2b [ANWW13] for the hash function, AES in counter mode for the encryption, HMAC for the PRFs, AEZ [HKR15] for the PRPs, a `/dev/random` seeded, AES-CTR based DRBG, and Elliptic Curve Multiset Hash [MSTA16] for the set hash function. Their implementation uses, whenever it is possible, SSE/AVX and AES-NI instructions set. A very small subset of the code uses the Boost library, and the counter mode is implemented from OpenSSL 1.0.2d. Excluding the test programs, we had to write around 5k code lines (*e.g.* this excludes the reference implementation of AEZ we used or Maitin-Shepard's implementation of ECMH).

Our prototype is entirely RAM-resident, and the client and server are implemented in a single process, in a single thread. Hence, the following evaluation do not account for disk-induced overhead (random access time) nor network-induced overhead ((de)serialization, latency, ...). Yet, we claim that it is representative from the point of view of computation time.

Data sets. For the evaluation of our implementation, we run tests on several subsets of the English Wikipedia, sizing between (around) 140M pairs (full data, 4.6M different keywords) and 140k pairs (10k different keywords), using stemming and after stop words removal. Each stem is treated as a keyword. The pre-processing step transformed the Wikipedia snapshot into reversed index stored as a JSON file. This file is parsed during the setup phase, using an event-driven parser, and encrypted as described in CXT's Setup algorithm.

Experimental Results. We ran experiments on a Intel Xeon E7 Nehalem CPU running at 2.66GHz with AES-NI and SSE4 instructions set. We recall that no multi-threading was used, and that the implementation is entirely sequential. Also, we omitted the communication induced overhead (instantaneous communications, infinite bandwith). When running on the largest dataset (140M document/keyword pairs), the program used up to 46GB of RAM.

To show scalability of the scheme, we executed several conjunctive search queries using two keywords. As in [CJJ+13] we noticed that the execution time of the query only depends on the number of document matching the first term of the query. Hence, in the

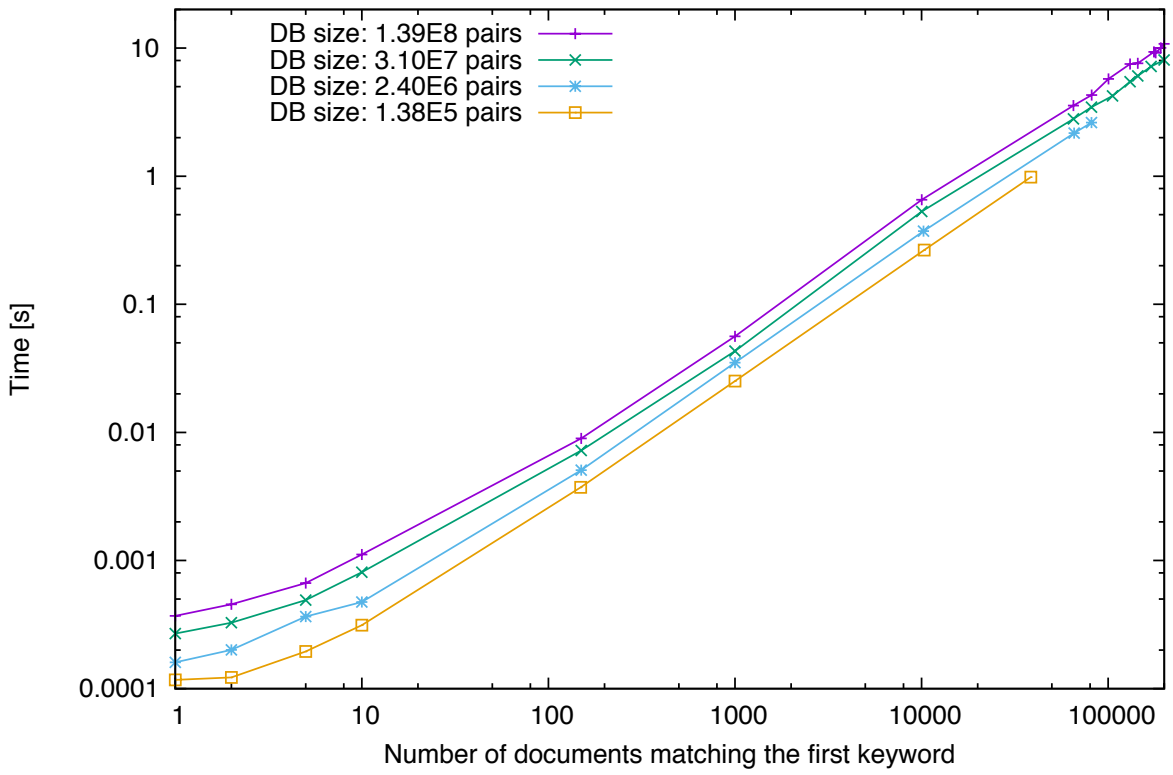


Figure 5.10: CXT's Search performance for two-keyword queries, with different databases.

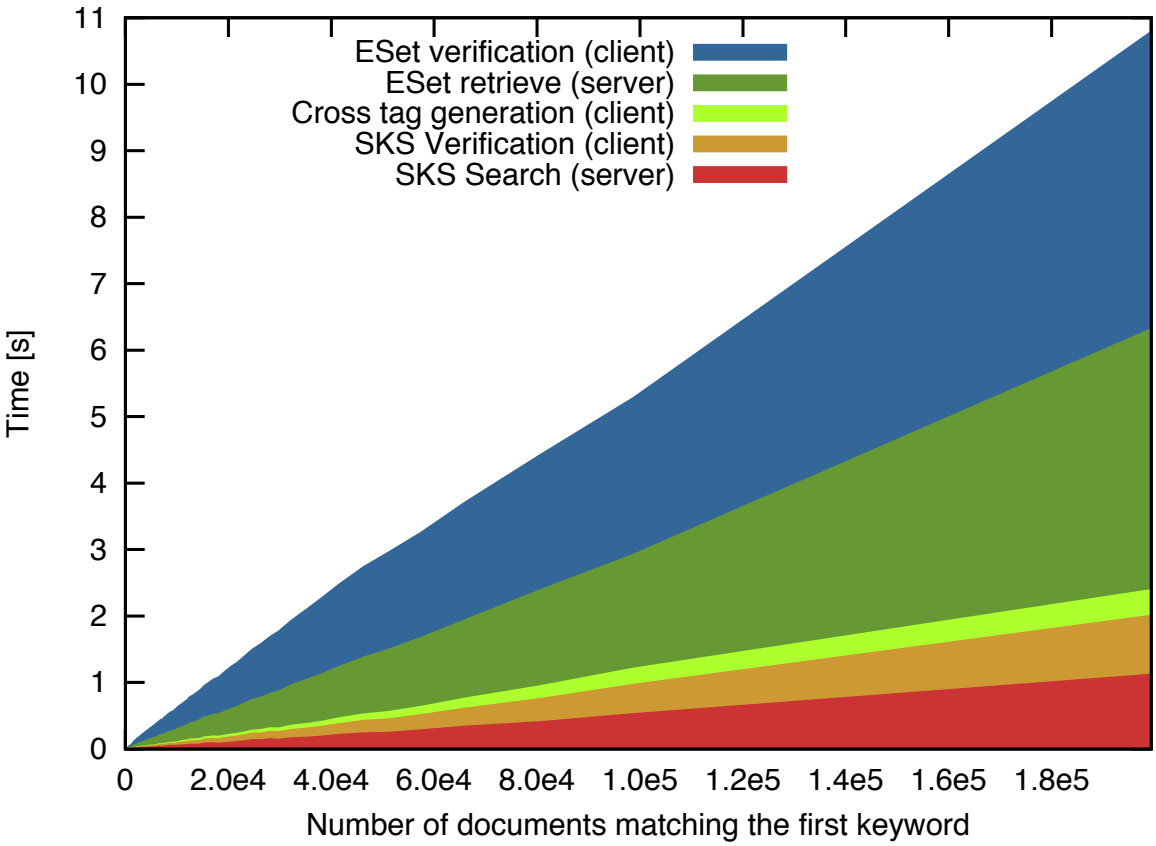


Figure 5.11: Repartition of the processing time of a two-keyword query. The graph is cumulative.

following graphs, the running time of a multiple-keyword query is given in function of the selectivity of the first keyword, as in Figure 5.10.

Figure 5.11 presents the contribution of the different parts of the protocol to CXT: we

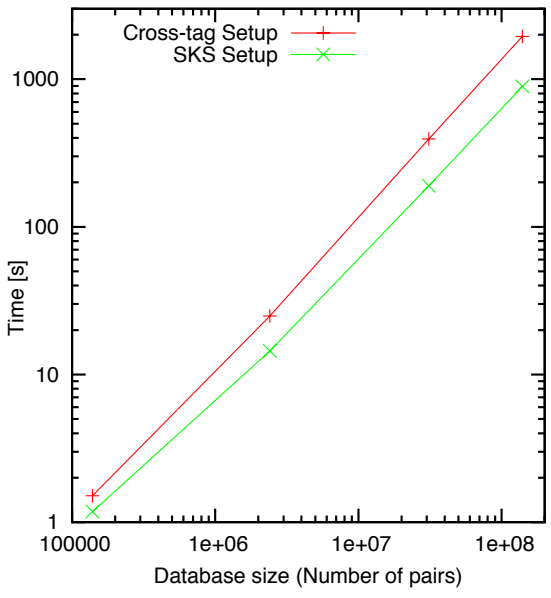


Figure 5.12: $\mathsf{CXT}_{\mathsf{bas}}$'s Setup performance.

split the single-keyword part (SKS) and the cross-tag part (ESet), and for these two, we separate the actual search/retrieve from the verification (and cross tag generation for the ESet). This figure gives a good insight on the verification cost: verification represents half of the running time, and even more if we consider that the need for verification implied non efficient choices for the E-Set data structure. Namely, in this case, one could have used a lot more efficient data structures for membership queries than the tree-based one we used, *e.g.* Bloom filters.

Figure 5.12 gives the encrypted database generation time for both the single-keyword scheme and the ESet containing the cross tags. Finally, Figure 5.13 presents the Update algorithm performance. We separated the case of a newly inserted keyword, as in the verifiable hash table instantiation we used, verifying the insertion of new element is more costly than verifying a value update.

Comparison with Existing Implementations. We can compare our evaluation results to the ones of Cash *et al.* [CJJ+13]. The main comparison point we have is the performance of the single-keyword search algorithm, without verification. But, as the implementation of [CJJ+13] is disk-resident, and despite the fact that concurrent accesses to RAID 5 disks are made, the cryptographic overhead is completely hidden by the I/O latency, and cryptographic computations can be overlapped with the disk accesses. This makes any direct comparisons complicated.

However, we can extrapolate the performance of a disk-based implementation of CXT using similar disk characteristic. First, we must notice that the verifications will not be affected by the use of disks: verification is done on the client side and can easily be performed in RAM, event for large result sets. Then, for the single-keyword part, as our instantiation is directly built on Cash *et al.* T-Set, we will inherit the same performances for the search itself. We will have to add the overhead needed by the tree-based verifiable

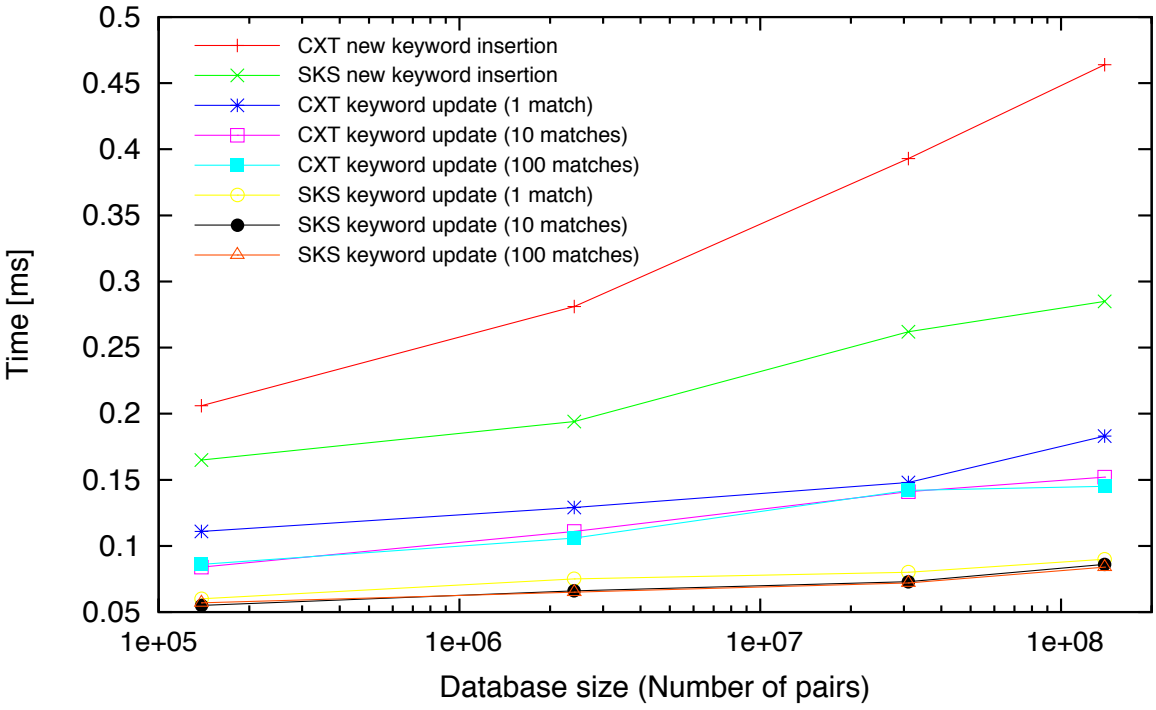


Figure 5.13: $\mathsf{CXT}_{\mathsf{bas}}$'s Update performance.

hash table. It might actually hold in RAM as it size is linear in the number of keywords in the database, but if we suppose that it also is disk-resident, given that our experiment shows that the tree spans over around 30 levels for 10,000 elements (the number of keywords in our largest dataset), less than 30 sequential random disk accesses would have to be performed for each query.

The E-Set access time will actually be the bottleneck of a disk-resident CXT implementation using our tree-based E-Set: for our largest dataset (139M pairs), every E-Set lookup will need at most 70 disk accesses. If we extrapolate [CJJ+13] estimates, this will take about 2 *seconds* per additional keyword when the first keyword matches 1,000 documents and 70 seconds per additional keyword when the first keyword matches 10,000 documents. We thus need a new technique to achieve verifiability for on-disk SSE.

Chapter 6

Conclusion

Real-Life Cryptography. Cryptography has been a key factor in driving globalization in recent years. It has increased the economic development and lowered impediments to communication between countries all over the globe. Cloud Computing has expanded this revolution enabling the growing of the computation power and availability.

In this work we contribute in the direction of securing existing Internet protocols and introducing new protocols for cloud computing environments.

Pseudorandom Number Generation. The cloud computing environment has enabled new attacks and threats that PRNG designers have no taken in consideration. This new setting requires new security models to analyze the security of a PRNG. Our contribution is door for the development of new work in the research and practical field. We showed that PRNGs based on old security models might break the whole security of cryptographic protocols because one of the most basic assumptions in these protocols is that there exists a source of real randomness. A Future work in this topic is the design of PRNGs using formal verification and automated tools that take in consideration complex rules like modern security models. A second open problem is the design of a secure PRNG using the SGX extensions [Gue16].

Password-Protected Secret Sharing. The development of outsourced data storage has missing one important security aspect which is the confidentiality of the data stored. Currently majors providers do not provide any system to store the data securely and they rely on each user to add the privacy layer missing by encrypting the data before uploading it. A secure and distributed password manager is a nice solution to store strong passwords of encrypted data. The introduction of Robust Gap Threshold Secret Sharing Scheme as underlying construction is interesting on its own, because it allows any secret sharing scheme to achieve robustness and use it as a black box.

Searchable Symmetric Encryption. With the explosion of data the problem of retrieving information has became trendy in the academy as well in the industry. Nevertheless, private information retrieval has still a long way to achieve the required efficiency to be usable in practice. In our work, we designed and implemented a Dynamic Searchable Symmetric Encryption that supports boolean queries with a reasonable throughput of 100,000 documents in 10 seconds. The problem of querying an encrypted data base is still an open problem, most of DSSE constructions that have been presented in the literature

so far come with several problems: Either they leak a significant amount of information or are inefficient in terms of space or search/update time. An open problem in SSE is the development of solutions that leak no information to the server that are not prohibitively expensive.

Bibliography

[AGV09] Adi Akavia, Shafi Goldwasser, and Vinod Vaikuntanathan. Simultaneous hardcore bits and cryptography against memory attacks. In Omer Reingold, editor, *TCC 2009*, volume 5444 of *LNCS*, pages 474–495. Springer, Heidelberg, March 2009. 20

[AK12] George Argyros and Aggelos Kiayias. I forgot your password: Randomness attacks against php applications. In *Proceedings of the 21st USENIX Conference on Security Symposium*, Security'12, pages 6–6, Berkeley, CA, USA, 2012. USENIX Association. 21

[And13] Some SecureRandom Thoughts, Aug 14st, 2013, 2013. http://android-developers.blogspot.fr/2013/08/some-securerandom-thoughts.html. 31

[ANWW13] Jean-Philippe Aumasson, Samuel Neves, Zooko Wilcox-O'Hearn, and Christian Winnerlein. BLAKE2: Simpler, smaller, fast as MD5. In Michael J. Jacobson Jr., Michael E. Locasto, Payman Mohassel, and Reihaneh Safavi-Naini, editors, *ACNS 13*, volume 7954 of *LNCS*, pages 119–135. Springer, Heidelberg, June 2013. 90

[BC16] Raphael Bost and Mario Cornejo. Implementation of some (Verifiable) SSE schemes. `http://gitlab.com/sse/sse_schemes`, 2016. Online. 90

[BFP16] Raphael Bost, Pierre-Alain Fouque, and David Pointcheval. Verifiable dynamic symmetric searchable encryption: Optimality and forward security. Cryptology ePrint Archive, Report 2016/062, 2016. `http://eprint.iacr.org/2016/062`. 72, 74, 75, 86, 87

[BH05] Boaz Barak and Shai Halevi. A model and architecture for pseudo-random generation with applications to /dev/random. In Vijayalakshmi Atluri, Catherine Meadows, and Ari Juels, editors, *ACM CCS 05*, pages 203–212. ACM Press, November 2005. 20, 23

[BJSL11] Ali Bagherzandi, Stanislaw Jarecki, Nitesh Saxena, and Yanbin Lu. Password-protected secret sharing. In Yan Chen, George Danezis, and Vitaly Shmatikov, editors, *ACM CCS 11*, pages 433–444. ACM Press, October 2011. 45, 46, 51

[Bla79] G. R. Blakley. Safeguarding cryptographic keys. *Proceedings of AFIPS 1979 National Computer Conference*, 48:313–317, 1979. 46

[BM84] G. R. Blakley and Catherine Meadows. Security of ramp schemes. In G. R. Blakley and David Chaum, editors, *CRYPTO'84*, volume 196 of *LNCS*, pages 242–268. Springer, Heidelberg, August 1984. 46

[BM92] Steven M. Bellovin and Michael Merritt. Encrypted key exchange: Password-based protocols secure against dictionary attacks. In *1992 IEEE Symposium on Security and Privacy*, pages 72–84. IEEE Computer Society Press, May 1992. 45

[BNPS03] Mihir Bellare, Chanathip Namprempre, David Pointcheval, and Michael Semanko. The one-more-RSA-inversion problems and the security of Chaum's blind signature scheme. *Journal of Cryptology*, 16(3):185–215, June 2003. 48

[Bou] The Bouncy Castle Crypto package is a Java implementation of cryptographic algorithms. http://www.bouncycastle.org/. 36

[BPR00] Mihir Bellare, David Pointcheval, and Phillip Rogaway. Authenticated key exchange secure against dictionary attacks. In Bart Preneel, editor, *EUROCRYPT 2000*, volume 1807 of *LNCS*, pages 139–155. Springer, Heidelberg, May 2000. 51, 69

[BPRW15] Allison Bishop, Valerio Pastro, Rajmohan Rajaraman, and Daniel Wichs. Essentially optimal robust secret sharing with maximal corruptions. Cryptology ePrint Archive, Report 2015/1032, 2015. http://eprint.iacr.org/2015/1032. 46

[BR93] Mihir Bellare and Phillip Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In V. Ashby, editor, *ACM CCS 93*, pages 62–73. ACM Press, November 1993. 13

[BR94] Mihir Bellare and Phillip Rogaway. Entity authentication and key distribution. In Douglas R. Stinson, editor, *CRYPTO'93*, volume 773 of *LNCS*, pages 232–249. Springer, Heidelberg, August 1994. 51

[BR95] Mihir Bellare and Phillip Rogaway. Provably secure session key distribution: The three party case. In *27th ACM STOC*, pages 57–66. ACM Press, May / June 1995. 51

[BR06] Mihir Bellare and Phillip Rogaway. The security of triple encryption and a framework for code-based game-playing proofs. In Serge Vaudenay, editor, *EUROCRYPT 2006*, volume 4004 of *LNCS*, pages 409–426. Springer, Heidelberg, May / June 2006. 17, 74

[BSV93] Carlo Blundo, Alfredo De Santis, and Ugo Vaccaro. Efficient sharing of many secrets. In *Proceedings of the 10th Annual Symposium on Theoretical Aspects of Computer Science*, STACS '93, pages 692–703, London, UK, UK, 1993. Springer-Verlag. 46

[Can01] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *42nd FOCS*, pages 136–145. IEEE Computer Society Press, October 2001. 46, 69

[CFOR12] Alfonso Cevallos, Serge Fehr, Rafail Ostrovsky, and Yuval Rabani. Unconditionally-secure robust secret sharing with compact shares. In David Pointcheval and Thomas Johansson, editors, *EUROCRYPT 2012*, volume 7237 of *LNCS*, pages 195–208. Springer, Heidelberg, April 2012. 49

[CGKO06] Reza Curtmola, Juan A. Garay, Seny Kamara, and Rafail Ostrovsky. Searchable symmetric encryption: improved definitions and efficient constructions. In Ari Juels, Rebecca N. Wright, and Sabrina De Capitani di Vimercati, editors, *ACM CCS 06*, pages 79–88. ACM Press, October / November 2006. 71, 72, 73, 74

[CGPR15] David Cash, Paul Grubbs, Jason Perry, and Thomas Ristenpart. Leakage-abuse attacks against searchable encryption. In Ray et al. [RLK15], pages 668–679. 72

[Che15] Mahdi Cheraghchi. Nearly optimal robust secret sharing. Cryptology ePrint Archive, Report 2015/951, 2015. http://eprint.iacr.org/2015/951. 46

[CHKO08] Philippe Camacho, Alejandro Hevia, Marcos A. Kiwi, and Roberto Opazo. Strong accumulators from collision-resistant hashing. In Tzong-Chen Wu, Chin-Laung Lei, Vincent Rijmen, and Der-Tsai Lee, editors, *ISC 2008*, volume 5222 of *LNCS*, pages 471–486. Springer, Heidelberg, September 2008. 88

[CJJ+13] David Cash, Stanislaw Jarecki, Charanjit S. Jutla, Hugo Krawczyk, Marcel-Catalin Rosu, and Michael Steiner. Highly-scalable searchable symmetric encryption with support for boolean queries. In Ran Canetti and Juan A. Garay, editors, *CRYPTO 2013, Part I*, volume 8042 of *LNCS*, pages 353–373. Springer, Heidelberg, August 2013. 13, 71, 74, 80, 82, 90, 91, 93, 94

[CJJ+14] David Cash, Joseph Jaeger, Stanislaw Jarecki, Charanjit S. Jutla, Hugo Krawczyk, Marcel-Catalin Rosu, and Michael Steiner. Dynamic searchable encryption in very-large databases: Data structures and implementation. In NDSS 2014 [NDS14]. 73, 74, 87

[CLLN14] Jan Camenisch, Anja Lehmann, Anna Lysyanskaya, and Gregory Neven. Memento: How to reconstruct your secrets from a single password in a hostile environment. In Garay and Gennaro [GG14], pages 256–275. 46

[CLN12] Jan Camenisch, Anna Lysyanskaya, and Gregory Neven. Practical yet universally composable two-server password-authenticated secret sharing. In Yu et al. [YDG12], pages 525–536. 46

[CLN15] Jan Camenisch, Anja Lehmann, and Gregory Neven. Optimal distributed password verification. In Ray et al. [RLK15], pages 182–194. 47

[CM05] Yan-Cheng Chang and Michael Mitzenmacher. Privacy preserving keyword searches on remote encrypted data. In John Ioannidis, Angelos Keromytis, and Moti Yung, editors, *ACNS 05*, volume 3531 of *LNCS*, pages 442–455. Springer, Heidelberg, June 2005. 71

[CS98] Ronald Cramer and Victor Shoup. A practical public key cryptosystem provably secure against adaptive chosen ciphertext attack. In Hugo Krawczyk, editor, *CRYPTO'98*, volume 1462 of *LNCS*, pages 13–25. Springer, Heidelberg, August 1998. 50

[CS99] Ronald Cramer and Victor Shoup. Signature schemes based on the strong RSA assumption. In *ACM CCS 99*, pages 46–51. ACM Press, November 1999. 70

[DGP09] Leo Dorrendorf, Zvi Gutterman, and Benny Pinkas. Cryptanalysis of the random number generator of the windows operating system. *ACM Trans. Inf. Syst. Secur.*, 13(1):10:1–10:32, November 2009. 21

[DH76] Whitfield Diffie and Martin E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 22(6):644–654, 1976. 10

[DHY02] Anand Desai, Alejandro Hevia, and Yiqun Lisa Yin. A practice-oriented treatment of pseudorandom number generators. In Lars R. Knudsen, editor, *EUROCRYPT 2002*, volume 2332 of *LNCS*, pages 368–383. Springer, Heidelberg, April / May 2002. 20, 23, 33, 36

[DIO98] Giovanni Di Crescenzo, Yuval Ishai, and Rafail Ostrovsky. Non-interactive and non-malleable commitment. In *30th ACM STOC*, pages 141–150. ACM Press, May 1998. 48

[DPR+13] Yevgeniy Dodis, David Pointcheval, Sylvain Ruhault, Damien Vergnaud, and Daniel Wichs. Security analysis of pseudo-random number generators with input: /dev/random is not robust. In Sadeghi et al. [SGY13], pages 647–658. 12, 20, 21, 22, 23, 24, 25, 26, 35, 40

[DSS00] Digital Signature Standard (DSS), FIPS PUB 186-2 with Change Notice. National Institute of Standards and Technology (NIST), FIPS PUB 186-2, U.S. Department of Commerce, January 2000. 33, 34, 36

[DSSW14] Yevgeniy Dodis, Adi Shamir, Noah Stephens-Davidowitz, and Daniel Wichs. How to eat your entropy and have it too - optimal recovery strategies for compromised RNGs. In Garay and Gennaro [GG14], pages 37–54. 20

[ElG85] Taher ElGamal. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Transactions on Information Theory*, 31:469–472, 1985. 13, 50

[Erl07] Úlfar Erlingsson. *Foundations of Security Analysis and Design IV: FOSAD 2006/2007 Tutorial Lectures*, chapter Low-Level Software Security: Attacks and Defenses, pages 92–134. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007. 19

[FIPR05] Michael J. Freedman, Yuval Ishai, Benny Pinkas, and Omer Reingold. Keyword search and oblivious pseudorandom functions. In Joe Kilian, editor, *TCC 2005*, volume 3378 of *LNCS*, pages 303–324. Springer, Heidelberg, February 2005. 50, 69

[FK00] Warwick Ford and Burton S. Kaliski, Jr. Server-assisted generation of a strong secret from a password. In *Proceedings of the 9th IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises*, pages 176–180, Washington, DC, USA, 2000. IEEE Computer Society. 45

[FSW03] Pierre-Alain Fouque, Jacques Stern, and Jan-Geert Wackers. Cryptocomputing with rationals. In Matt Blaze, editor, *FC 2002*, volume 2357 of *LNCS*, pages 136–146. Springer, Heidelberg, March 2003. 55

[FVK+15] Ben A. Fisch, Binh Vo, Fernando Krell, Abishek Kumarasubramanian, Vladimir Kolesnikov, Tal Malkin, and Steven M. Bellovin. Malicious-client security in blind seer: A scalable private DBMS. In *2015 IEEE Symposium on Security and Privacy*, pages 395–410. IEEE Computer Society Press, May 2015. 72

[GG14] Juan A. Garay and Rosario Gennaro, editors. *CRYPTO 2014, Part II*, volume 8617 of *LNCS*. Springer, Heidelberg, August 2014. 98, 99

[GGM86] Oded Goldreich, Shafi Goldwasser, and Silvio Micali. How to construct random functions. *Journal of the ACM*, 33(4):792–807, October 1986. 50

[GMP15] Sanjam Garg, Payman Mohassel, and Charalampos Papamanthou. TWORAM: Round-optimal oblivious RAM with applications to searchable encryption. Cryptology ePrint Archive, Report 2015/1010, 2015. http://eprint.iacr.org/2015/1010. 87, 90

[Goh03] Eu-Jin Goh. Secure indexes. Cryptology ePrint Archive, Report 2003/216, 2003. http://eprint.iacr.org/2003/216. 71

[Gol04] Oded Goldreich. *Foundations of cryptography.* Cambridge University Press, 2004. 72, 74, 77

[GPR06] Zvi Gutterman, Benny Pinkas, and Tzachy Reinman. Analysis of the linux random number generator. In *2006 IEEE Symposium on Security and Privacy*, pages 371–385. IEEE Computer Society Press, May 2006. 21

[GSW04] Philippe Golle, Jessica Staddon, and Brent R. Waters. Secure conjunctive keyword search over encrypted data. In Markus Jakobsson, Moti Yung, and Jianying Zhou, editors, *ACNS 04*, volume 3089 of *LNCS*, pages 31–45. Springer, Heidelberg, June 2004. 71

[Gue16] Shay Gueron. A memory encryption engine suitable for general purpose processors. Cryptology ePrint Archive, Report 2016/204, 2016. http://eprint.iacr.org/2016/204. 95

[Gut98] Peter Gutmann. Software generation of practically strong random numbers. In *Proceedings of the 7th Conference on USENIX Security Symposium - Volume 7*, SSYM'98, pages 19–19, Berkeley, CA, USA, 1998. USENIX Association. 20, 27

[HDWH12] Nadia Heninger, Zakir Durumeric, Eric Wustrow, and J. Alex Halderman. Mining your ps and qs: Detection of widespread weak keys in network devices. In *Proceedings of the 21st USENIX Conference on Security Symposium*, Security'12, pages 35–35, Berkeley, CA, USA, 2012. USENIX Association. 21

[Hea14] The Heartbleed Bug, 2014. http://heartbleed.com. 19, 41

[HKR15] Viet Tung Hoang, Ted Krovetz, and Phillip Rogaway. Robust authenticated-encryption AEZ and the problem that it solves. In Elisabeth Oswald and Marc Fischlin, editors, *EUROCRYPT 2015, Part I*, volume 9056 of *LNCS*, pages 15–44. Springer, Heidelberg, April 2015. 90

[HPS08] Jeffrey Hoffstein, Jill Pipher, and J.H. Silverman. *An Introduction to Mathematical Cryptography.* Springer Publishing Company, Incorporated, 1 edition, 2008. 15

[IBM14] Recent Fixes in IBM SecureRandom, 2014. http://www.cigital.com/justice-league-blog/2014/05/06/recent-fixes-ibmsecurerandom/. 38

[IIES14] G. Irazoqui, M. S. Inci, T. Eisenbarth, and B. Sunar. Fine grain cross-vm attacks on xen and vmware. In *Big Data and Cloud Computing (BdCloud), 2014 IEEE Fourth International Conference on*, pages 737–744, Dec 2014. 19

[IKK12] Mohammad Saiful Islam, Mehmet Kuzu, and Murat Kantarcioglu. Access pattern disclosure on searchable encryption: Ramification, attack and mitigation. In *NDSS 2012*. The Internet Society, February 2012. 72

[IKNP03] Yuval Ishai, Joe Kilian, Kobbi Nissim, and Erez Petrank. Extending oblivious transfers efficiently. In Dan Boneh, editor, *CRYPTO 2003*, volume 2729 of *LNCS*, pages 145–161. Springer, Heidelberg, August 2003. 69

[ILL89] Russell Impagliazzo, Leonid A. Levin, and Michael Luby. Pseudo-random generation from one-way functions (extended abstracts). In *21st ACM STOC*, pages 12–24. ACM Press, May 1989. 21

[Jab01] David P. Jablon. Password authentication using multiple servers. In David Naccache, editor, *CT-RSA 2001*, volume 2020 of *LNCS*, pages 344–360. Springer, Heidelberg, April 2001. 45

[JD] Java Decompiler project. http://jd.benow.ca. 42

[JDP] Java Platform Debugger Architecture (JPDA). http://docs.oracle.com. 41

[JKK14] Stanislaw Jarecki, Aggelos Kiayias, and Hugo Krawczyk. Round-optimal password-protected secret sharing and T-PAKE in the password-only model. In Palash Sarkar and Tetsu Iwata, editors, *ASIACRYPT 2014, Part II*, volume 8874 of *LNCS*, pages 233–253. Springer, Heidelberg, December 2014. 12, 13, 47, 48, 51, 53, 54, 57, 58, 69, 70

[JKKX16] Stanislaw Jarecki, Aggelos Kiayias, Hugo Krawczyk, and Jiayu Xu. Highly-Efficient and Composable Password-Protected Secret Sharing. Cryptology ePrint Archive, Report 2016/144, 2016. http://eprint.iacr.org/. 13, 47, 58, 69

[JS13] Mahabir Prasad Jhanwar and Reihaneh Safavi-Naini. Unconditionally-secure robust secret sharing with minimum share size. In Ahmad-Reza Sadeghi, editor, *FC 2013*, volume 7859 of *LNCS*, pages 96–110. Springer, Heidelberg, April 2013. 49

[Kal97] Burton S. Kaliski, Jr. Ieee p1363: A standard for rsa, diffie-hellman, and elliptic-curve cryptography (abstract). In *Proceedings of the International Workshop on Security Protocols*, pages 117–118, London, UK, UK, 1997. Springer-Verlag. 31

[KHL13] Soo Hyeon Kim, Daewan Han, and Dong Hoon Lee. Predictability of android OpenSSL's pseudo random number generator. In Sadeghi et al. [SGY13], pages 659–668. 21

[KO12] Kaoru Kurosawa and Yasuhiro Ohtaki. UC-secure searchable symmetric encryption. In Angelos D. Keromytis, editor, *FC 2012*, volume 7397 of *LNCS*, pages 285–298. Springer, Heidelberg, February / March 2012. 72, 74, 75

[KO13] Kaoru Kurosawa and Yasuhiro Ohtaki. How to update documents verifiably in searchable symmetric encryption. In Michel Abdalla, Cristina Nita-Rotaru, and Ricardo Dahab, editors, *CANS 13*, volume 8257 of *LNCS*, pages 309–328. Springer, Heidelberg, November 2013. 75, 86, 87

[KPR12] Seny Kamara, Charalampos Papamanthou, and Tom Roeder. Dynamic searchable symmetric encryption. In Yu et al. [YDG12], pages 965–976. 87

[KR01] Joe Kilian and Phillip Rogaway. How to protect DES against exhaustive key search (an analysis of DESX). *Journal of Cryptology*, 14(1):17–35, 2001. 17

[KSWH98] John Kelsey, Bruce Schneier, David Wagner, and Chris Hall. Cryptanalytic attacks on pseudorandom number generators. In Serge Vaudenay, editor, *FSE'98*, volume 1372 of *LNCS*, pages 168–188. Springer, Heidelberg, March 1998. 20, 33, 36

[LP14] Allison Bishop Lewko and Valerio Pastro. Robust secret sharing schemes against local adversaries. Cryptology ePrint Archive, Report 2014/909, 2014. http://eprint.iacr.org/2014/909. 49

[MJ96] Keith M. Martin and W.-A. Jackson. "a combinatorial interpretation of ramp schemes". *Australasian Journal of Combinatorics*, 14:51–60, 1996. 46

[MMS13] Kai Michaelis, Christopher Meyer, and Jörg Schwenk. Randomly failed! The state of randomness in current java implementations. In Ed Dawson, editor, *CT-RSA 2013*, volume 7779 of *LNCS*, pages 129–144. Springer, Heidelberg, February / March 2013. 21, 24, 31, 36

[MS81] R. J. McEliece and D. V. Sarwate. On sharing secrets and reed-solomon codes. *Commun. ACM*, 24(9):583–584, September 1981. 46, 49

[MSTA16] Jeremy Maitin-Shepard, Mehdi Tibouchi, and Diego F. Aranha. Elliptic Curve Multiset Hash. arXiv, Report 1601.06502, 2016. http://arxiv.org/abs/1601.06502. 87, 90

[NDS14] *NDSS 2014*. The Internet Society, February 2014. 98, 102

[NR97] Moni Naor and Omer Reingold. Number-theoretic constructions of efficient pseudo-random functions. In *38th FOCS*, pages 458–467. IEEE Computer Society Press, October 1997. 13, 58, 64

[Oba11] Satoshi Obana. Almost optimum t-cheater identifiable secret sharing schemes. In Kenneth G. Paterson, editor, *EUROCRYPT 2011*, volume 6632 of *LNCS*, pages 284–302. Springer, Heidelberg, May 2011. 49

[OP01] Tatsuaki Okamoto and David Pointcheval. The gap-problems: A new class of problems for the security of cryptographic schemes. In Kwangjo Kim, editor, *PKC 2001*, volume 1992 of *LNCS*, pages 104–118. Springer, Heidelberg, February 2001. 48

[Ope13] OpenSSL PRNG Is Not (Really) Fork-safe, Aug 21st, 2013. http://emboss.github.io/blog/2013/08/21/openssl-prng-is-not-really-fork-safe/. 29

[Orc] Orchid is a Tor client implementation and library written in pure Java. http://www.subgraph.com/orchid.html. 42

[Pai99] Pascal Paillier. Public-key cryptosystems based on composite degree residuosity classes. In Jacques Stern, editor, *EUROCRYPT'99*, volume 1592 of *LNCS*, pages 223–238. Springer, Heidelberg, May 1999. 13, 70

[PTT09] Charalampos Papamanthou, Roberto Tamassia, and Nikos Triandopoulos. Cryptographic accumulators for authenticated hash tables. Cryptology ePrint Archive, Report 2009/625, 2009. http://eprint.iacr.org/2009/625. 87

[RFY+13] L. Ren, C. Fletcher, X. Yu, M. van Dijk, and S. Devadas. Integrity verification for path oblivious ram. In *Proceedings of the 17th IEEE High Performance Extreme Computing Conference*, September 2013. 87, 88, 90

[RLK15] Indrajit Ray, Ninghui Li, and Christopher Kruegel:, editors. *ACM CCS 15*. ACM Press, October 2015. 98

[RSA78] Ronald L. Rivest, Adi Shamir, and Leonard M. Adleman. A method for obtaining digital signature and public-key cryptosystems. *Communications of the Association for Computing Machinery*, 21(2):120–126, 1978. 10

[RTSS09] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In Ehab Al-Shaer, Somesh Jha, and Angelos D. Keromytis, editors, *ACM CCS 09*, pages 199–212. ACM Press, November 2009. 19

[SGY13] Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors. *ACM CCS 13*. ACM Press, November 2013. 99, 101

[Sha49] C. Shannon. Communication theory of secrecy systems. *Bell System Technical Journal, Vol 28, pp. 656–715*, Oktober 1949. 10

[Sha79] Adi Shamir. How to share a secret. *Communications of the Association for Computing Machinery*, 22(11):612–613, November 1979. 46, 54

[SHA95] Secure hash standard. National Institute of Standards and Technology, NIST FIPS PUB 180-1, U.S. Department of Commerce, April 1995. 24, 32, 33

[Sho01] Victor Shoup. OAEP reconsidered. In Joe Kilian, editor, *CRYPTO 2001*, volume 2139 of *LNCS*, pages 239–259. Springer, Heidelberg, August 2001. 17

[SPS14] Emil Stefanov, Charalampos Papamanthou, and Elaine Shi. Practical dynamic searchable encryption with small leakage. In NDSS 2014 [NDS14]. 71, 72, 75

[SPWS13] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. SoK: Eternal war in memory. In *2013 IEEE Symposium on Security and Privacy*, pages 48–62. IEEE Computer Society Press, May 2013. 41

[SWP00] Dawn Xiaodong Song, David Wagner, and Adrian Perrig. Practical techniques for searches on encrypted data. In *2000 IEEE Symposium on Security and Privacy*, pages 44–55. IEEE Computer Society Press, May 2000. 71

[Tor] The Tor Project. https://www.torproject.org. 42

[Tre01] Luca Trevisan. Extractors and pseudorandom generators. *J. ACM*, 48(4):860–879, 2001. 24

[TT05] Roberto Tamassia and Nikos Triandopoulos. Computational bounds on hierarchical data processing with applications to information security. In Luís Caires, Giuseppe F. Italiano, Luís Monteiro, Catuscia Palamidessi, and Moti Yung, editors, *ICALP 2005*, volume 3580 of *LNCS*, pages 153–165. Springer, Heidelberg, July 2005. 87

[TW88] Martin Tompa and Heather Woll. How to share a secret with cheaters. *Journal of Cryptology*, 1(2):133–138, 1988. 49

[vdVdSCB12] Victor van der Veen, Nitish dutt Sharma, Lorenzo Cavallaro, and Herbert Bos. Memory Errors: The Past, the Present, and the Future. In *Proceedings of the 15th International Symposium on Research in Attacks, Intrusions, and Defenses (RAID)*, September 2012. 19

[VNT61] John Von Neumann and A.H. Taub. *John von Neumann : collected works. vol. 5 : design of computers, theory of automata and numerical analysis*. Pergamon, Oxford, 1961. Reprinted : 1976. 19

[WNL+14] Xiao Shaun Wang, Kartik Nayak, Chang Liu, T.-H. Hubert Chan, Elaine Shi, Emil Stefanov, and Yan Huang. Oblivious data structures. In Gail-Joon Ahn, Moti Yung, and Ninghui Li, editors, *ACM CCS 14*, pages 215–226. ACM Press, November 2014. 88, 90

[YDG12] Ting Yu, George Danezis, and Virgil D. Gligor, editors. *ACM CCS 12*. ACM Press, October 2012. 98, 101

[YHCL15] Xun Yi, Feng Hao, Liqun Chen, and Joseph K. Liu. Practical threshold password-authenticated secret sharing protocol. In Günther Pernul, Peter Y. A. Ryan, and Edgar R. Weippl, editors, *ESORICS 2015, Part I*, volume 9326 of *LNCS*, pages 347–365. Springer, Heidelberg, September 2015. 46

Résumé

La cryptographie a été un facteur clé pour permettre la vente de services et du commerce par Internet. Le *cloud computing* a amplifié cette révolution et est devenu un service très demandé grâce à ses avantages comme : puissance de calcul importante, services à bas coûts, rendement, évolutivité, accessibilité et disponibilité. Parallèlement à la hausse de nouveaux business, des protocoles pour des calculs sécurisés ont aussi émergé.

Le but de cette thèse est de contribuer à la sécurité des protocoles d'Internet existants en fournissant une analyse de la source aléatoire de ces protocoles et en introduisant des protocoles mieux adaptés pour les environnements des *cloud computing*. Nous proposons de nouvelles constructions en améliorant l'efficacité des solutions actuelles afin de les rendre plus accessibles et pratiques. Nous fournissons une analyse de sécurité détaillée pour chaque schéma avec des hypothèses raisonnables.

Nous étudions la sécurité du *cloud computing* à différents niveaux. D'une part, nous formalisons un cadre pour analyser quelques-uns des générateurs de nombres pseudo-aléatoires populaires à ce jour qui sont utilisés dans presque chaque application cryptographique. D'autre part, nous proposons deux approches efficaces pour des calculs en cloud. Le premier permet à un utilisateur de partager publiquement son secret de haute entropie avec des serveurs différents pour plus tard le récupérer par interaction avec certains de ces serveurs en utilisant seulement son mot de passe et sans données authentifiées. Le second permet à un client d'externaliser à un serveur une base de données en toute sécurité, qui peut être recherchée et modifiée ultérieurement.

Mots Clés

Cryptographie, Protocoles d'Internet, Générateurs de nombres pseudo-aléatoires, Cloud computing, Base de données, Secret partagé

Abstract

Cryptography has been a key factor in enabling services and products trading over the Internet. Cloud computing has expanded this revolution and it has become a highly demanded service or utility due to the advantages of high computing power, cheap cost of services, high performance, scalability, accessibility as well as availability. Along with the rise of new businesses, protocols for secure computation have as well emerged.

The goal of this thesis is to contribute in the direction of securing existing Internet protocols by providing an analysis of the sources of randomness of these protocols and to introduce better protocols for cloud computing environments. We propose new constructions, improving the efficiency of current solutions in order to make them more accessible and practical. We provide a detailed security analysis for each scheme under reasonable assumptions.

We study the security in a cloud computing environment in different levels. On one hand, we formalize a framework to study some popular real-life pseudorandom number generators used in almost every cryptographic application. On the other, we propose two efficient applications for cloud computing. The first allows a user to publicly share its high-entropy secret across different servers and to later recover it by interacting with some of these servers using only his password without requiring any authenticated data. The second, allows a client to securely outsource to a server an encrypted database that can be searched and modified later.

Keywords

Cryptography, Internet protocols, Pseudorandom number generator, Cloud computing, Databases, Secret sharing