

Aalto University
School of Science
Master's Programme in Computer, Communication and Information Sciences

Kirthivaasan Puniamurthy

A proof viewer for State-separating proofs: Yao's Garbling Scheme

Master's Thesis
Espoo, December 31, 2020

Supervisors: Professor Christopher Brzuska
Dr. Alla Levina
Advisors: Konrad Kohbrok M.Sc. (Tech.), Aalto University
Sabine Oechsner Phd. (Tech), Aarhus University
Christoph Egger M.Sc. (Tech.), Friedrich-Alexander-
Universität Erlangen-Nürnberg (FAU)

Aalto University
School of Science

Master's Programme in Computer, Communication and
Information Sciences

ABSTRACT OF
MASTER'S THESIS

Author:	Kirthivaasan Puniamurthy		
Title:	A proof viewer for State-separating proofs: Yao's Garbling Scheme		
Date:	December 31, 2020	Pages:	75
Major:	Computer Science	Code:	SCI3042
Supervisors:	Professor Christopher Brzuska Dr. Alla Levina		
Advisors:	Konrad Kohbrok M.Sc. (Tech.), Aalto University Sabine Oechsner Phd. (Tech), Aarhus University Christoph Egger M.Sc. (Tech.), Friedrich-Alexander- Universität Erlangen-Nürnberg (FAU)		
<p>Cryptographic proofs are the most important aspect of reasoning about a protocol/construction's security. Hence, <i>effective</i> proof communication is a natural and important goal for cryptographers. In this work, we focus on using the state separating proofs (SSP) framework for proving the security of cryptographic protocols. The central cryptographic construction we study throughout this work is Yao's garbling scheme, a way to achieve secure multi-party computation (MPC). Complex cryptographic protocols such as Yao's garbling scheme, tend to have long proofs. Moreover, SSPs tend to be detailed and visual in nature. Hence, our primary objective is the creation of an <i>SSP proof viewer</i>, a suitable medium for viewing SSP proofs. Our second objective is to present the security Yao's garbling scheme effectively. Thirdly, we aim to find properties and methods of effective proof communication in the field of cryptography. Lastly we test our ideas on proof communication and the effectiveness of the proof viewer, weconduct a small user study on a group of cryptography course participants. The results of this study suggests that our ideas of proof communication and the proof viewer are useful and effective.</p>			
Keywords:	garbled circuits, state separating proofs, proof understanding and communication, proof visualization		
Language:	English		

Acknowledgements

First and foremost I would like to express my deepest gratitude to Chris Brzuska for being a wonderful mentor, and for always encouraging me to learn and understand, and most of all for having faith in me. Being advised by Chris has been an absolute joy, and has taught me so many things. I am also grateful to him for teaching me cryptography, a subject that I had always wanted to learn.

I would like to thank Konrad Kohbrok for helping me (especially during the initial stages of this thesis) with understanding F^* , and verification in general. I learnt a lot from those sessions and meetings, and derived immense pleasure from seeing how problems can be approached systematically. I would also like to thank Christoph Egger for the many interesting and thought provoking conversations which has enriched my understanding of cryptography. Much of Chapters 6-7 is a direct result of our discussions. It was a joy to discuss many crypto-related topics (as well as non-crypto topics). I also wish to thank Sabine Oechsner for answering several of my garbled circuits related queries and for providing me with excellent resources for studying them, including the papers which she co-authored, which were an invaluable resource for this work. I would also like to thank Jan Winkelmann for spending time with me on a few occasions to share his thoughts on SSP visualization, and some practical aspects of the proof viewer. I would like to extend my thanks also to the course participants who participated in the user study on garbling schemes, and for being merciful and kind despite the organizational quirks.

Last but not least I would like to express my sincere thanks to friends who had to endure hearing about the thesis/garbled circuits/proof viewer, and who had to be guinea pigs for testing these ideas! I am very grateful to my family, friends and loved ones, (you all know who you are), for your help, unwavering support and for making me a better person.

Espoo, December 31, 2020

Kirthivaasan Puniamurthy

Abbreviations and Acronyms

IND-CPA	Indistinguishability against chosen plaintext attack
SSP	State separating proof
SSC	State separating construction
MPC	Multi party computation
2PC	2-party computation
OT	Oblivious Transfer
GC	Garbled Circuit
DAG	Directed Acyclic Graph
PoC	Proof of Concept

Contents

Abbreviations and Acronyms	4
1 Introduction	8
1.1 Cryptographic proofs of security	8
1.2 Computer-aided cryptography	10
1.3 State-separating proofs	10
1.4 Research objectives	11
1.5 Content of thesis	11
2 Computational Security Notions	13
2.1 Adversarial power	14
2.1.1 Oracles	14
2.1.2 Computationally bounded adversaries	15
2.1.3 Security parameter of cryptosystems	15
2.2 Game-based security	16
2.2.1 Indistinguishability under Chosen-Plaintext Attack (IND-CPA) security	16
2.3 Simulation-based security	18
2.3.1 Semantic security	19
2.4 Reductions in cryptography	19
2.5 Simulation in MPC	21
3 State separating proofs of cryptographic security	23
3.1 Foundations of state separating proofs	24
3.1.1 Packages	24
3.1.2 Games in SSP	24
3.2 Formulation of IND-CPA in SSP	25
3.3 Code equivalence	26
3.3.1 Inlining	27
3.3.2 Code transformation	27
3.3.3 Example: Code equivalence	27

3.4	Package composition	28
3.4.1	Sequential composition	28
3.4.2	Parallel composition	29
3.5	Semantic security in SSP	29
3.6	Proof example: Semantic security and IND-CPA equivalence .	31
3.7	State separating constructions (SSC)	33
3.8	Reductions in SSP	33
3.9	Conventions and style	34
4	Garbled Circuits	35
4.1	Secure multi-party computation	35
4.2	Yao's garbling scheme	36
4.2.1	Gate garbling	37
4.2.2	Circuit representation	38
4.2.3	Circuit garbling	39
4.2.4	Garbled circuit evaluation	40
4.3	Oblivious Transfer	41
4.3.1	Simple OT	42
4.4	MPC from garbled circuits	42
4.5	Garbled circuits optimizations	44
5	Security of Garbled Circuits	47
5.1	BHR syntax	47
5.2	Correctness of garbling schemes	48
5.3	Security of garbling schemes	50
6	Proof verification and understanding	52
6.1	Cryptographic proof verification	52
6.1.1	Formal methods and proof verification	53
6.1.2	Limitations of machine proof verification	54
6.2	Proof communication	54
6.3	Information hiding	54
6.4	Proof structure	55
6.5	Intent	55
6.5.1	Naming	56
6.5.2	SSP graph layouts	57
6.6	Related work	58
7	Proof viewer for SSPs	59
7.1	Proof viewing and flow	59
7.2	Implementation details	62

7.2.1	Overview of design goals and choices	62
7.3	User study	64
7.3.1	Setup and questions	65
7.3.2	Interpretation of results	66
7.4	Future work	66
7.5	Conclusion	68
A	Attack on deterministic encryption scheme	73
B	User study	74
B.1	Questions	74
B.1.1	Structural questions	74
B.1.2	Technical questions	74
B.1.3	Subjective questions	75

Chapter 1

Introduction

Cryptography can sometimes be referred to as real life magic. This is because cryptography is literally the art of achieving impossible things. For example, zero knowledge should not be possible. In fact, more generally, multi-party computation (MPC) should not be possible! Even if we restrict ourselves to Minicrypt¹ [19], deceptively simple primitives such as one-way functions (OWFs) can be used for all sort of incredible things, such as signatures [23]².

Cryptography also has an answer to ethical questions of user data privacy and computation - namely the field of secure multi-party computation (MPC). MPC offers the possibility of two or more parties to jointly compute some function (e.g. machine classification, electronic voting, secure auctions) and to obtain the result *without* any of the parties having to reveal their inputs. Garbling schemes (first introduced by Yao [35]) are an elegant and practical solution for MPC and have seen a long line of optimizations and usage in the real world (more recently in the area of secure machine learning [3], [25], [27]). This work is centered around understanding *Yao's garbling scheme*, and its cryptographic *proof* of security.

1.1 Cryptographic proofs of security

A cryptographic construction must be *proven* to be secure. The way this is done is by modeling the construction mathematically within a logical framework. The idea of reasoning *mathematically* with a program or protocol is a big step in research development - and not something that is practiced in all

¹A world where one-way functions exist (Many believe that Minicrypt exists unlike Impagliazzo's other worlds)

²In recent times, signature schemes based on OWFs are seeing a resurgence due to post-quantum cryptography.

areas of (cyber) security.

Cryptographic designers, even very experienced ones, can easily come up with a protocol/construction that they deem intuitively secure and might convince others of the security of their construction - yet, we have rich literature on broken constructions, e.g., the Needham-Schröder protocol [26], the Bleichenbacher attack [10]. The difference in cryptography is that assumptions are made explicit, and intuition is substituted by rigorous proof. A rigorous proof that is almost *mechanical* to the point of being verifiable by a machine (we will get back to this point a bit later). Formal proofs that are rigorous and verifiable are crucial in cryptography, since we want to make guarantees of security that are absolutely watertight based on clear assumptions on what an adversary can or cannot do (also known as *adversarial capacity*) when interacting with a cryptosystem. Note that proofs only provide guarantees to the extent that they are covered by the model, but model gaps are easier to spot and understand than protocol gaps.

A proof then convinces us via a series of proof steps, that the cryptosystem satisfies certain specific, desired properties such as correctness (i.e. the cryptosystem does what we want it to do) and security (i.e. it achieves it by preserving some notion of security such as confidentiality/integrity). Hence the proof makes it clearer to us (the people writing, as well as those trying to verify and understand the proof), that the construction is secure and functional, without a doubt. But is this really the case?

The problem is, countless times people have produced even *proofs* of cryptographic constructions, that have bugs in them. Moreover, these bugs may or may not be apparent and can be buried deep - making them hard to find, especially if people have already accepted the constructions in real world protocols (this could be confirmation bias, or to support legacy systems or even just plain obstinacy)³. A bogus proof is perhaps even more insidious than the bogus construction itself because it argues that something must be true (i.e. the construction must be secure) by presenting a fake argument.

Another problem is, are proofs even clear to begin with? Do we understand why something is true even if a correct, verifiable proof is available? Technically yes, but some mathematicians confess that despite having verified a valid proof, their understanding of the problem still has not been deepened by the proof. Moreover, it (usually) takes a lot of time and effort to parse a correct proof (for human verifiers). This is why keeping in mind that proofs are a *social means* - a method of communication about truth, helps us come up with proofs that are clearer, better perceived and understood.

³However, there have been efforts to overcome this. A recent example of such an effort is the complete re-design of the Transport Layer Security protocol (TLS 1.3).

1.2 Computer-aided cryptography

Over many years the cryptographic community has invested and placed much thought on improving methods for writing and verifying proofs with the assistance of machines [17]. The problem of manual proof writing has been addressed before, most notably in Halevi’s (provocative) paper [17], in which he writes about the lack of such tools and a possible approaches to build them.

Many of the tools from computer-aided cryptography have been used successfully in different areas such as verified implementations and automatic (security) verification. However, tools/programs for effective cryptographic *proof writing* (with machine assistance) are rather limited, which we discuss further in Section 6.1.1.

We find that an important element of tools for humans, is to have an effective interface between human and computer. For example, the Tamarin tool for symbolic verification [29] has a visual display of proofs, while the EasyCrypt tools has a nice front-end⁴ (Proof General).

1.3 State-separating proofs

Many wonderful innovations in thinking about cryptography have sprung out of pedagogical needs as well, for example, Rosulek’s fantastic book *The Joy of Cryptography* [32] (which was initially developed for an undergraduate course in cryptography), and constructive cryptography by Maurer [28] (with notation and visualizations which the author claims was motivated to simplify concepts of compositional security for undergraduate students as well). We find that the human aspect has been a driving force for many of the approaches for structuring and writing cryptographic proofs. In our examples we find that both frameworks (with Rosulek’s being an *informal* framework), employ interesting and elegant methods for cryptographic reasoning.

State-separating proofs by Brzuska, Delignat-Lavaud, Fournet, Kohbrok and Kohlweiss (BDFKK [11]) come from a similar place, to make cryptographic proofs more suitable for (automatic) verification, but more importantly, arising from the need for better proof communication and understanding.

⁴In Emacs.

1.4 Research objectives

While formal verification of cryptographic proofs are useful, they contain far too many details to be easily accessible to humans. Moreover, they may not provide insight into *why* a proof is true, and where the significant/interesting parts of a proof are. While proof writers learn a lot, it is less clear if proof readers learn as much. It would be useful to instead have an appropriate proof medium for cryptographic proofs where the proof reader is not only able to efficiently verify a proof but is able to learn and gain understanding of the proof. To this end we define the following research goals.

1. To build a proof viewer for SSPs.
2. To present the security proof of Yao's garbling scheme.
3. To find properties and methods of effective (cryptographic) proof communication.
4. To test our ideas of effective (cryptographic) proof communication empirically.

1.5 Content of thesis

In Chapter 2 we introduce security paradigms in cryptography and define computational security notions for symmetric encryption (namely IND-CPA and semantic security). We also discuss proof methods in cryptography, especially reductions.

Chapter 3 presents the state separating proofs (SSP) framework for cryptographic proofs. Most of our discussions in other chapters will draw upon concepts and definitions found within this chapter. Especially, the contents in Chapter 3 will be used in Chapter 5-?? which present the definitions and proof of security of Yao's garbling scheme, and Chapter 7 (proof viewer for SSPs).

Next, we switch modes to understanding the concept of garbled circuits (Chapter 4). We build an intuition for how garbled circuits are constructed, evaluated and used for secure multi-party computation (MPC). Then, we briefly discuss garbled circuits optimizations and their impact on proofs of security.

In Chapter 5, we begin by introducing the syntax for garbling schemes and the subsequent definitions of correctness for garbling schemes (in terms

of packages). Then we define the *security* of garbling schemes in the style of SSPs.

Chapter 6 is devoted to proof verification and understanding, and how the two concepts differ and how they are alike. Concretely, we provide ways to effectively communicate cryptographic proofs via effective use of *information hiding proof structure*, and *intent*.

Finally in Chapter 7 we discuss the design, features and implementation of a proof viewer for SSPs. In this chapter, we discuss how we implemented the concepts for effective proof communication such as information hiding and providing proof structure in the proof viewer. We also provide certain important implementation details, i.e. the way proofs are represented in the viewer and how graphs are generated and edited. We then present the results of a user study we conducted on a small group of course participants, which tests the ideas we hypothesized in Chapter 6 and Chapter 7.

Chapter 2

Computational Security Notions

The first step in designing a cryptosystem is to define its functionality or behaviour, also known as the cryptosystem's *correctness*. For example, (informally) the correctness of an encryption scheme might state that the decryption of a ciphertext always yields the original message that was encrypted. Jumping ahead to the example of garbling schemes, their correctness requires that given a circuit C and its garbling \tilde{C} , the evaluation of C on an input x yields the same output as evaluating x on its garbling, i.e. for all x , $C(x) = \tilde{C}(x)$.

The next step is to define the *security* of the cryptosystem, by specifying a set of security properties that the system shall satisfy. A basic but important security property that we might want from a scheme is message confidentiality. In this chapter, we introduce security definitions for IND-CPA secure encryption schemes, and in later chapters see how they are used as building blocks for more complex MPC protocols, e.g. garbled circuits.

We first begin our discussion with the concept of adversarial power (Section 2.1) and ways to model them, which will then lead us to the two main classes of security notions (Sections 2.2 and Section 2.3). Within these classes, we present computational notions for symmetric encryption schemes, an essential cryptographic primitive used in the vast majority of cryptographic protocols. Subsequently, in Section 2.4 we introduce a fundamental proof technique in cryptography, called *reductions*. Finally, we end the chapter with a brief discussion on simulation-based notions with regard to MPC protocols (Section 2.5).

2.1 Adversarial power

In cryptography, we model adversaries as algorithms that interact with a cryptosystem or primitive. Typically, we think of an adversary as belonging to a certain *class* of algorithms (Section 2.1.2).

To formalize an adversary's interaction with a cryptosystem, we set up a system (which we also refer to as a *game*, defined in Section 2.2) that functions as a wrapper for our cryptosystem or cryptographic primitive. The system provides an adversary with an *interface* that it can interact with or query. In particular, the system's interface specifies a set of functions commonly known as its *oracles*, which it gives access to an adversary in order to break the system's security. Note that an adversary may combine oracle calls in any way that it chooses, but with the constraint that these calls/queries are bounded. The bound on the number of oracle calls an adversary is allowed to make is typically asymptotically defined, e.g. by a polynomial in the *security parameter* of the cryptosystem (definition of security parameter can be found in Section 2.1.3). We now move on to understanding the concept of *oracles* which is central to cryptography and complexity theory.

2.1.1 Oracles

Oracles are functions that live in a system (jumping ahead, this "system" is also known as a *package* Section 3.1.1) and (possibly) share state among other oracles in the same system. An oracle is commonly considered to be a *black-box* function¹, meaning that its input-output behaviour is defined, but an adversary cannot observe its internal computations, i.e. its state and execution. Furthermore, oracles are assumed to be efficient, i.e. they are PPT algorithms, and hence must terminate.

Since oracles are functions that compute, they usually are described in some computational model such as Turing machines. However, we choose to use *pseudocode* to describe oracles, for several reasons, (such as to show functional equivalence and to enable a well-defined concept of composability) which we will discuss later (Section 3.3). Oracle procedures may define and allocate *local* variables that are not shared by other oracles within the same system in order to perform computations. Additionally, oracles are not allowed to make recursive calls to themselves, avoiding issues of infinite loops/scheduling. An example of an oracle definition is the ENC oracle, which

¹In some cases, the existence of an oracle is purely hypothetical, but useful for reasoning about the consistency of a system of complexity classes.

can be seen in Figure 2.1. Notice that in the case of ENC oracle, the state that it operates on is essentially the key k that is used in encryption.

2.1.2 Computationally bounded adversaries

A landmark achievement in cryptography was Shannon’s One-Time Pad (OTP) encryption for message privacy [33], which he proved to be secure against computationally unbounded adversaries. OTP achieves perfect secrecy, whereby the best strategy an adversary can adopt in order to break the system’s security (confidentiality) is to try and guess the key that was used in encryption by brute force. While powerful, information theoretic constructions such as OTP are not useful in practice due to their requirement of large key size.

Hence, in order to create more practical systems, most modern cryptographic protocols are designed to be secure against adversaries that are instead *computationally bounded*. The notion of perfect secrecy is relaxed by constraining adversaries to be efficient, i.e. adversaries are modeled as probabilistic polynomial time (PPT) algorithms. As a consequence, keys may be of fixed length and significantly shorter than messages.

Another problem with the One-Time Pad (as the name indicates) is that it requires a fresh key on every encryption, which is impractical in most cases. Hence, cryptographers introduce the notion of *probabilistic* encryption schemes, whereby encryption is randomized, i.e. the encryption algorithm produces ciphertexts that are different on each encryption, in contrast to deterministic schemes.

2.1.3 Security parameter of cryptosystems

A cryptosystem’s level of security may be defined or parameterized by a security parameter that when fixed describes the complexity of breaking the system (e.g. for keys of size 128 bits, an adversary has to try 2^{128} many keys). The security parameters of cryptographic schemes almost always correspond to the size of the keys used, which are most commonly instantiated with powers of two (e.g. AES-128, SHA-256). In this work, we assume the convention of denoting the security parameter by λ .

The security parameter is typically defined asymptotically and is often implicit in the definition of a cryptosystem. The benefit of reasoning asymptotically with security, is not having to assign any particular value to λ . An asymptotic definition of security also allows us to formalize the notion of “upgrading” a scheme’s security i.e. by increasing/doubling a concrete instantiation of λ .

Next in Section 2.2 and Section 2.3, we introduce the two main paradigms of security in cryptography with probabilistic encryption as an illustrating example.

2.2 Game-based security

The idea of game-based security definitions is to allow an adversary to interact with a real and ideal system or *game*² to try to distinguish between them. The real game is set up to use the cryptographic construction we intend to prove secure, whereas the ideal game is unbreakable by design. Intuitively, distinguishing real and ideal games with non-negligible probability shows that the adversary has learnt some information or weakness of the real system, making it insecure in contrast to the ideal system. Recall that an adversary interacts with a game by calling the oracles that it provides through its interface. Hence, the interfaces of both ideal and real games must be identical, otherwise an adversary can trivially distinguish between them.

Games are given names e.g. G , that are typically super-scripted by a single bit b to indicate whether it is the real game G^0 or the ideal game G^1 . An adversary \mathcal{A} playing one of these games, denoted by $\mathcal{A} \rightarrow G^b$ for some game G^b where $b \in \{0, 1\}$, outputs 0 if it believes it is in G^0 and 1 if it believes it is in G^1 . The probability with which an adversary is able to distinguish real and ideal games is known as the adversary's *advantage*. Then, a cryptosystem is said to be secure (under some game-based notion) if for all PPT adversaries \mathcal{A} , the advantage of \mathcal{A} , with respect to real and ideal games G^0, G^1 , is negligible. Below we provide the definition of adversarial advantage and negligible functions.

Definition 2.1 (Advantage). The advantage of an adversary that can distinguish between a real game G^0 and an ideal game G^1 is defined as

$$\text{Adv}(\mathcal{A}; G^0, G^1) := |\Pr[1 = \mathcal{A} \rightarrow G^0] - \Pr[1 = \mathcal{A} \rightarrow G^1]|$$

Definition 2.2 (Negligible function). A negligible function $\text{negl} : \mathbb{N} \rightarrow [0, 1]$ tends to zero faster than any positive inverse polynomial $\frac{1}{p(n)}$.

2.2.1 Indistinguishability under Chosen-Plaintext Attack (IND-CPA) security

IND-CPA security is a game-based notion meant to capture the security of probabilistic encryption schemes, specifically message (or plaintext) confi-

²The term “system” is sometimes used to refer to “game” in this work

dentiality. In fact, IND-CPA gives an even stronger notion of confidentiality, that is it allows an adversary to choose the messages that are to be encrypted, but is unable to distinguish between encryptions.

We now describe a game (see Figure 2.1) for defining IND-CPA security of a probabilistic symmetric encryption scheme se . Note that since encryption is randomized, we use the symbol $\leftarrow_{\$}$ instead of \leftarrow to denote that the ciphertext is produced by a randomized algorithm. In 2.1, we see that both real and ideal games each provide an adversary with access to a single oracle, namely ENC.

Gind-cpa_{se}^0	Gind-cpa_{se}^1
$\text{ENC}(m)$	$\text{ENC}(m)$
if $k = \perp$	if $k = \perp$
$k \leftarrow_{\$} \{0, 1\}^\lambda$	$k \leftarrow_{\$} \{0, 1\}^\lambda$
$c \leftarrow_{\$} se.enc(k, m)$	$c \leftarrow_{\$} se.enc(k, 0^{ m })$
return c	return c

Figure 2.1: Definition of games for IND-CPA secure encryption scheme se .

Roughly speaking, the definition of IND-CPA states that an efficient adversary cannot learn anything from the ciphertext, except for its length. The intuition behind why this definition is meaningful is, if an adversary chooses a message m and yet is unable to differentiate between the encryptions of m and $0^{|m|}$, then really nothing can be deduced from observing the ciphertext alone. This “flavour” of IND-CPA is called Real-Zeroes IND-CPA (abbreviated RoZ-CPA), because an adversary is unable to distinguish between an encryption of its message and zeroes of the same length.

Note that, in the definition of IND-CPA below we have included the interface that Gind-cpa provides to \mathcal{A} , in this case the interface simply being the set containing a single oracle ENC which we denote by placing above the arrows, i.e. $\xrightarrow{\text{ENC}}$.

Definition 2.3 (IND-CPA). An encryption scheme se is considered IND-CPA secure if for all PPT adversaries \mathcal{A} the games Gind-cpa_{se}^0 and Gind-cpa_{se}^1 are indistinguishable i.e.

$$\Pr[1 = \mathcal{A} \xrightarrow{\text{ENC}} \text{Gind-cpa}_{se}^0] - \Pr[1 = \mathcal{A} \xrightarrow{\text{ENC}} \text{Gind-cpa}_{se}^1] \leq \text{negl}(\lambda)$$

Notice that this definition encrypting the zeroes is rather arbitrary - we could have picked a random string, or ones. A fact which we will not prove

here is that there are several IND-CPA notions that are all equivalent, including a “weird” IND-CPA notion which we will use in garbled circuits (Section ??).

2.3 Simulation-based security

Simulation-based security is a paradigm for security definitions, frequently used in defining the security of Secure multi-party (MPC) protocols (more on this in Section 2.5) and in frameworks that facilitate composition, such as Universal Composability (UC) [13].

In the simulation paradigm there are two worlds; a real world, which represents the adversary’s view of the real protocol, and an ideal world which operates according to some ideal functionality, together with an additional entity known as a *simulator*.

Before delving into the details of what a simulator is, we should first briefly discuss the meaning of “view” and the concept of ideal functionality. The view of a party is all of the information that it is able to observe during the execution of a protocol, also referred to as the protocol’s *transcript*, as well as knowledge of its own inputs (if it is a participant of the protocol) or the inputs of a participant it has corrupted. On the other hand, the ideal functionality of a system is the mechanism that implements the correctness of the system, given minimal information (e.g. the length of a message instead of the message itself), and achieves security in a way that is unbreakable by definition (e.g. by using a trusted third party). We show an example of ideal functionality is defined in simulation-based definitions of MPC protocols in Section 2.5.

The purpose of the simulator is to generate a view that is indistinguishable from an adversary’s view of the real world, based on minimal information. If the real view is indistinguishable from the ideal view except negligibly, we say that the cryptosystem is secure. Put another way, if an adversary is able to distinguish the real view from a simulated view, it means that it hasn’t learnt anything useful from the interaction/transcript, since it could have simulated this interaction itself.

In our view, “worlds” in the simulation paradigm are modeled as games - the only difference being that the ideal “world” or game is parameterized by a simulator that is separate from the code of the system, unlike game-based definitions, which one can think of as having hard-coded simulators. This connection allows us to use simulation-based notions in code based games proofs³.

³We see an example of using simulators in game based security definitions in chapters

2.3.1 Semantic security

Semantic security is essentially a simulation-based analog of IND-CPA security for probabilistic encryption schemes. The game setup is as follows. An adversary is allowed to query the system with a message m and receive its encryption. In the real world or game, the adversary receives an encryption of m , whereas in the ideal world it receives a *simulated* ciphertext, based only on the length of the message.

Gsem-sec_{se}^0	$\text{Gsem-sec}_{se, \mathcal{S}}^1$	$\mathcal{S}(\ell)$
$\text{ENC}(m)$	$\text{ENC}(m)$	if $k = \perp$
if $k = \perp$	$c \leftarrow \mathcal{S}(0^{ m })$	$k \leftarrow_{\$} \{0, 1\}^\lambda$
$k \leftarrow_{\$} \{0, 1\}^\lambda$	return c	junk $\leftarrow 0^{ \ell }$
$c \leftarrow_{\$} se.enc(k, m)$		$c \leftarrow_{\$} se.enc(k, \text{junk})$
return c		return c

Figure 2.2: Definition of real and ideal worlds semantic security for the encryption scheme se . On the extreme right is the simulator algorithm \mathcal{S} .

Definition 2.4 (Semantic security). An encryption scheme is considered semantically secure if there exists a PPT simulator \mathcal{S} such that for all PPT adversaries \mathcal{A} , the games Gsem-sec^0 and $\text{Gsem-sec}_{\mathcal{S}}^1$ are indistinguishable, i.e.

For completeness, we include the definition of a simulator \mathcal{S} that satisfies the definition of semantic security. \mathcal{S} is specified as encrypting “junk”, which is just a string of zeroes. The simulator \mathcal{S} could have also just encrypted the unary encoding of the length (ℓ), but using a separate variable **junk** makes it clearer that the simulator could have encrypted any arbitrary string. In this case **junk** is set to zeroes 0^ℓ , since we would ultimately like to show the connection with the Real-or-Zeroes IND-CPA definition we presented earlier.

We see that the definition of semantic security is quite similar to the definition IND-CPA secure probabilistic encryption, and in particular we will prove that these two notions are in fact equivalent, in the Chapter 3.

2.4 Reductions in cryptography

Cryptographic constructions and protocols usually rely on the security of certain cryptographic primitives or assumptions of well known hard problems (e.g. the hardness of factoring large numbers or the discrete logarithm

2 and in the proof of Yao’s garbling scheme

problem). A reduction is a proof technique, used abundantly in cryptography for reasoning about the security of larger systems that depend on secure primitives/systems or hardness assumptions.

More concretely, let C be a complex system that is constructed using a primitive or smaller system P that is secure (either by assumption or proven to be secure). To prove C secure we assume there exists an efficient hypothetical adversary \mathcal{A}_C that is able to break the complex system C . Assuming such \mathcal{A}_C exists, we show that using \mathcal{A}_C , we are able to construct an efficient adversary \mathcal{A}_P that breaks the security of primitive P through an efficient transformation known as a *reduction*. At this point, we have essentially derived a contradiction, since P is secure by definition, proving that there cannot exist a successful, efficient adversary for C .

As an example, let us prove via reduction that appending λ many zeroes to the end of a message before encryption with se is still IND-CPA secure. To construct our new “padding” encryption scheme, we reuse the encryption oracle ENC of the original IND-CPA games Gind-cpa^b , which we instead call with a padded message m' .

```

Pad.ENC( $m$ )
-----
 $m' \leftarrow m \parallel 0^\lambda$ 
return ENC( $m'$ ) // ENC of Gind-cpa

```

Figure 2.3: The Pad encryption scheme

Example 2.5 (Pad encryption is IND-CPA secure). Let us assume for the purpose of contradiction that there exists an adversary \mathcal{A}_{Pad} that breaks Pad. We show that we can transform \mathcal{A}_{Pad} via reduction \mathcal{R} into an adversary \mathcal{A}_{se} that breaks se .

$\mathcal{R}(m)$	\mathcal{A}_{se}
$m' \leftarrow m \parallel 0^\lambda$	1 : \dots // \mathcal{A}_{Pad} preparing message m
$c \leftarrow \text{ENC}(m')$ // ENC of Gind-cpa	2 : $m \leftarrow \diamond$
return c	3 : $c \leftarrow \mathcal{R}(m)$
	4 : \dots // \mathcal{A}_{Pad} attack using c
	5 : \dots
	6 : return b // distinguishing bit

Figure 2.4: Reduction \mathcal{R} (left) and sketch of adversary \mathcal{A}_{se} (right)

Since the adversary \mathcal{A}_{pad} is black-box, we do not know its code, but we may roughly reason about it in phases, one for preparing the message query, and another performing the distinguishing attack. The reduction is invoked each time \mathcal{A}_{se} makes an ENC call. Intuitively, the reduction “intercepts” the ENC call of \mathcal{A}_{se} and redirects it with a call to the ENC of the original Gind-cpa game, after preprocessing the message by appending 0^λ .

The code of the reduction is very similar (in fact, in our example it is identical) to the code of the Pad system. This is no coincidence, since the reduction can be thought of as the parts of the more complex system excluding the primitive which we want to reduce to. In a sense, the reduction functions as a converter, translating messages from the hypothetical adversary to the primitive we want to break.

2.5 Simulation in MPC

The simulation paradigm is often used in multi-party computation (MPC), because it is more convenient to define the ideal functionality of such protocols (both correctness and security). Moreover, since multiple parties engage in the protocol, and participants themselves may be malicious, it is simpler to formulate security in terms of the *views* of these parties. Moreover, certain MPC protocols are not symmetric, i.e. not all parties perform the same actions.

As an example let us consider a protocol for securely computing the logical AND (Figure 2.5) of multiple parties. Each party A, B and C provides AND with their respective inputs b_A, b_B, b_C . The AND node then computes the logical AND of their inputs and broadcasts the result to all parties, and serves as the ideal functionality of this protocol. Therefore, the AND node performing the computation is referred to as a *trusted third party*, because the other parties trust that it will only perform the logical AND computation and nothing else, without modifying the inputs or output. Many MPC protocols follow a similar pattern when formulating definitions for the ideal world of their protocol.

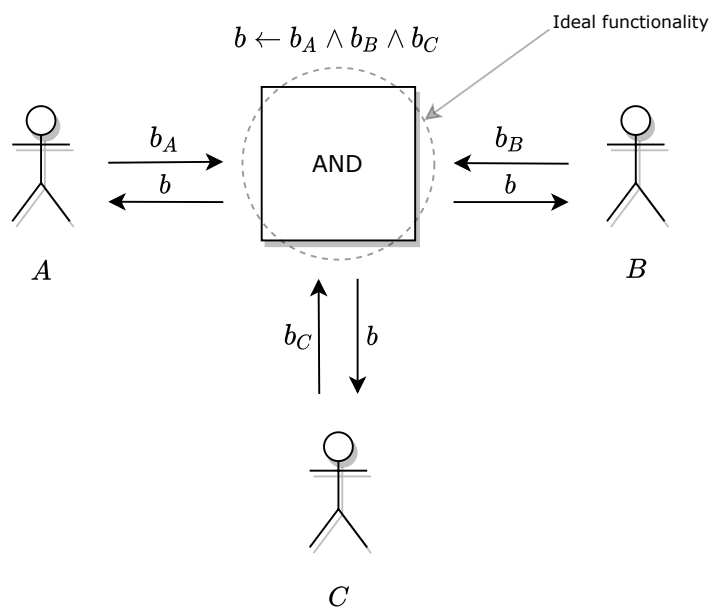


Figure 2.5: Secure AND protocol

Chapter 3

State separating proofs of cryptographic security

Cryptographic game hopping has long been used in proving protocol security, first introduced by Shoup [34], and then refined into code-based game-playing by Bellare, Rogaway [7] to prove triple encryption security. One could argue that this approach was pioneered due to necessity, because of the increasing complexity of such cryptographic protocols. Moreover, proofs in the code based games framework tend to be clearer, as they use pseudocode as a computational model, instead of Turing machines.

State separating proofs (SSP), is a cryptographic proof framework for code based games introduced by Brzuska, Delignat-Lavaud, Fournet, Kohbrok and Kohlweiss (BDFKK [11]). The SSP methodology takes the code based games approach further by focusing on modularity and composition in proofs via state separation, in addition to introducing techniques for effective proof communication and understanding through visualization.

This chapter is organized as follows. We first introduce the main objects of state separating proofs, i.e. packages and games (Section 3.1.1 and Section 3.1.2). Then, we introduce the formulation of IND-CPA in SSP (Section 3.2), and use this to further explain code equivalence (Section 3.3) and package composition (Section 3.4). We then present a proof for the equivalence of IND-CPA and semantic security, within the SSP framework (Section 3.6). This proof not only serves as an example of SSP-style proofs, reductions and code equivalences but also shows how we use simulation notions in games. Next, we provide a discussion on state separating *constructions* (Section 3.7) and reductions in SSP (Section 3.8). The chapter ends with a brief explanation of some conventions we follow in our SSP-style proofs (Section 3.9).

3.1 Foundations of state separating proofs

3.1.1 Packages

State separating proofs (SSPs) are built from computational units called *packages*. A package maintains state ¹, and provide a set of oracles that may be called by other packages through its interface. A package P 's outer interface is known as its *oracles*, sometimes denoted by $\text{out}(P)$, or more intuitively $[\rightarrow P]$. On the other hand, the set of calls that P makes to other packages, is called its *dependencies*, denoted by $\text{in}(P)$ or $[P \rightarrow]$. In this work we will prefer using the arrow notation as it avoids confusion between an “outer” interface and an “inner” interface’. Note that these interfaces are encoded by call graphs (for example in Figure 3.1) which define exactly which oracles a package may call, since it is important that packages do not automatically get access to another package’s oracles unless specified.

An important property of packages is that their state is only accesible to oracles that are within that package and not accessible from external packages, hence the term *state separation*. The only way state variables can be obtained from a package P is if a caller package C queries an oracle of P that returns (a subset of) the state variables explicitly (e.g. by a GET oracle), or returns it indirectly as the result of some computation . Additionally, packages may be paramaterized by algorithms/schemes or even other packages.

As we will see later, the concept of packages can be naturally extended to enable composition, with the application of certain operations on packages (Section 3.4). This way, packages can themselves be defined in a modular way, i.e. by defining it to be a composition of smaller packages.

3.1.2 Games in SSP

Recall that cryptographic games are systems that are interacted with by an adversary, through a game’s interface. In the SSP framework, we define games as packages with no dependencies, i.e. a package G is a game if $[G \rightarrow] = \emptyset$. Note adversaries in SSP are also *packages*, and which interact with games². More specifically, adversaries are packages that when called (by an implicit run oracle), return a distinguishing bit after interacting with a game, i.e. its guess whether it interacted with a real or idea game.

¹Packages are similar to *libraries* in Mike Rosulek’s *The Joy of Cryptography*

²Since we mostly deal with hypothetical adversaries, these are “black-box” packages in SSP, i.e. they have an interface, oracles, but we cannot know their implementation or behaviour.

Another consequence of viewing games as packages is that games may also be modular (since packages can be modular, recall that these simply state separating constructions (SSCs)). An example of a modular game can be found in our modular definition of IND-CPA (Figure 3.1). Thinking of games as packages not only facilitates modularity, but composition. Since composition is defined on packages, we immediately obtain the useful notion of *game composition*. Concretely, this allows us to compose different security notions (that are formulated as games) together. This also gives us the useful property of multi-instance game composition which we discuss further in Section 3.4.

3.2 Formulation of IND-CPA in SSP

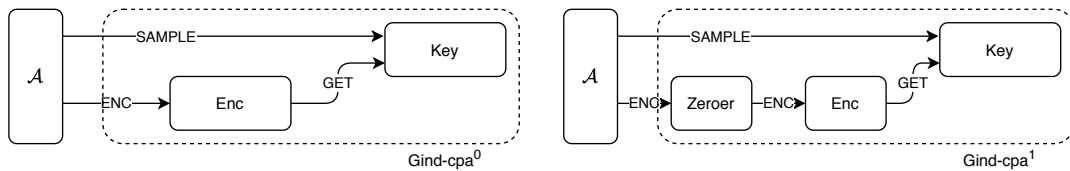


Figure 3.1: Definition of IND-CPA secure encryption scheme se . $G_{\text{ind-cpa}}^0$ (real game on the left) and $G_{\text{ind-cpa}}^1$ (ideal game on the right)

<u>Key</u>	<u>Zeroer</u>	<u>Enc</u>
Parameters	$\text{ENC}(m)$	$\text{ENC}(m)$
λ : sec. parameter	$c \leftarrow \text{ENC}(0^{ m })$	$k \leftarrow \text{GET}()$
	return c	$c \leftarrow \$ se.\text{enc}(k, m)$
Package State		return c
k : key		
<u>SAMPLE()</u>	<u>GET()</u>	
assert $k = \perp$	assert $k \neq \perp$	
$k \leftarrow \$ \{0, 1\}^\lambda$	return k	
return ()		

Figure 3.2: Oracles of the Key package (left) and ENC oracles of Zeroer and Enc packages (right)

Definition 3.1 (IND-CPA secure encryption scheme se). A probabilistic encryption scheme se is secure if for all PPT adversaries \mathcal{A}

$$\text{Adv}(\mathcal{A} \rightarrow \text{Gind-cpa}^0, \mathcal{A} \rightarrow \text{Gind-cpa}^1)$$

is negligible.

In Figure 3.1, we have a graphical representation of an SSP formulation of modular IND-CPA (modular, since the games are not composed of single monolithic packages). The graphs are essentially call graphs, with packages are visually represented as rectangles, and oracle calls from one package to another drawn with arrows, with their names as labels. What would have been the monolithic counterparts of these games are drawn as dotted rectangles around the other two packages (labelled Gind-cpa^0 and Gind-cpa^1 , with the same names as the IND-CPA games in Chapter 2).

In Figure 3.2 we define the oracles for the packages that the games Gind-cpa^0 and Gind-cpa^1 consist of. We see that the only difference between them is the introduction of a **Zeroer** package in the ideal game. Another difference, is we allow an adversary to sample a key for encryption at the beginning of the game, which does not alter the definition in any significant way. Note that we have also included the adversary \mathcal{A} which we will omit, due to reasons explained in Section 3.9.

The IND-CPA game proceeds as follows. After sampling a key, the adversary is allowed to choose a message and submit it for encryption by calling **ENC**. The order of the calls are and are enforced by assert statements. In this game, **SAMPLE** must be called before **ENC**, since **ENC** obtains the key via **GET** - which in turn requires that the key already been generated (**assert** $k \neq \perp$). In the ideal game adversary's message m is first converted into a string of zeroes $0^{|m|}$ of the same length by the **Zeroer** package before being encrypted by **Enc**.

A somewhat obvious point to note is that the adversary package \mathcal{A} is not allowed complete access to emphall of the oracles provided by the **Key** package (as specified by the graph), otherwise it could trivially break the system by obtaining the key via **GET**.

We will return to IND-CPA security in Section 3.6 where we will show that IND-CPA security is equivalent to semantic security and now turn to composition and proof techniques.

3.3 Code equivalence

The idea of code equivalence in SSP is to determine if two packages have identical or *perfectly indistinguishable* behaviour. The first step in checking

for code equivalence of two packages Pkg1 and Pkg2 , is to ensure that they implement the same interfaces, that is their set of oracles and dependencies should be identical, i.e. $[\rightarrow \text{Pkg1}] = [\rightarrow \text{Pkg2}]$ and $[\text{Pkg1} \rightarrow] = [\text{Pkg2} \rightarrow]$. The fact that a package’s interfaces matches another is sometimes called a *syntactical argument*, since we can reason about swapping one package for another without checking anything else.

3.3.1 Inlining

The next step in establishing code equivalence is to prove that Pkg1 ’s oracles and dependencies (and the dependencies of its dependencies and so on, i.e. recursively) are *functionally equivalent* to Pkg2 ’s oracles and dependencies. The standard way to do this is via *inlining*, i.e. by rewriting oracle calls by their definitions, with its variables substituted by the arguments it was called with, where necessary. Often, variable name collisions may occur, and these can be fixed by renaming them. Once we have “expanded out” the function calls of an oracle by inlining, we may still have to perform some *code transformation* steps, since the oracle’s behaviour might be functionally equivalent, but not identical in code description.

3.3.2 Code transformation

To perform code transformation, certain lines or branches of code may be removed entirely (if they are not reachable), or certain operators and functions may be manipulated to shape them into a description (in pseudocode) that is identical. Another useful technique in code transformation is to isolate what is invariant in an algorithm or a part of code, and to argue that manipulating certain parts of the algorithm does not affect the invariant, e.g. an algorithm that enumerates through a list to save its elements into a *set* does not need to be concerned with the *order* in which it enumerates the list, because the set is inherently unordered (the preserved invariant). Naturally, code equivalence is an important proof step in SSPs but can be tedious and easy to get wrong when done manually. Ways of performing code equivalence using formal methods will be discussed further in Section 6.1.1 of Chapter 6.

3.3.3 Example: Code equivalence

As an example, let us consider the modular version of IND-CPA depicted in Figure 3.1, which we claim to be code equivalent to the monolithic games we defined in Chapter 2 (Section 2.1). Technically, they are not *entirely* code

equivalent since we have provided an additional **SAMPLE** oracle in the IND-CPA formulation here, but we restrict our focus to showing code equivalence of the only significant difference between the real and ideal games, which is the **ENC** oracle. Additionally, we will focus on showing that the modular ideal game Gind-cpa_{se}^1 is code equivalent to its monolithic counterpart. The steps are shown in Figure 3.3. Notice that this particular case is quite clean, since we only had to perform inlining and no additional code transformation steps.

<u>Zeroer</u>	<u>Zeroer \circ Enc</u>	<u>Zeroer \circ Enc \circ Key</u>
ENC (m)	ENC (m)	ENC (m)
$c \leftarrow \text{ENC}(0^{ m })$ return c	$k \leftarrow \text{GET}$ $c' \leftarrow \$se.enc(k, 0^{ m })$ $c \leftarrow c'$ return c	assert $k \neq \perp$ $c \leftarrow \$se.enc(k, 0^{ m })$ return c
		SAMPLE ()
		assert $k = \perp$ $k \leftarrow \$\{0, 1\}^\lambda$

Figure 3.3: Code equivalence of Gind-cpa^1 with modular Gind-cpa^1 by inlining

3.4 Package composition

SSP packages have an algebra for package composition, which the interested reader may refer to in the original BDFKK [11] paper for a formal and comprehensive treatment. Below we informally describe the two types of package composition, namely, sequential and parallel composition, and provide an intuition for how they are used in practice.

3.4.1 Sequential composition

Our IND-CPA call graph depicts packages that are connected by oracle calls, essentially exhibiting a *sequential* composition of packages. To illustrate the concept of package composition in SSP, we take the example of Gind-cpa^1 , in which we have a sequential composition of **Zeroer**, **Enc** and **Key** packages. We call this sequential composition, since **Enc** makes an oracle call to **Key**, which returns an output to **Enc**, indicating a sequence/order of calls. The reader

might also intuit that this *sequential* package composition is associative. It doesn't matter if we consider either $(\text{Zeroer} \rightarrow \text{Enc}) \rightarrow \text{Key}$ or $\text{Zeroer} \rightarrow (\text{Enc} \rightarrow \text{Key})$, since in both cases calls are made in the order $\text{Zeroer} \rightarrow \text{Enc} \rightarrow \text{Key}$. Since SSP call graphs have the structure of a directed acyclic graph (DAG), we tend to visualize this (and with it package composition as well), *horizontally*, i.e. from left to right, like in the IND-CPA formulation above.

3.4.2 Parallel composition

Packages also have a concept of parallel composition, which is especially useful for the composition of multi-instance games. Often protocols combine multiple security notions in their construction, making it necessary for separate security game to be proven for each one of these notions. Certain protocols may even have a self similar structure whereby a security notion is applied multiple times (e.g. garbled circuits). In contrast to sequential composition, we may think of parallel composition as growing *vertically*, by stacking games from top to bottom. The goal of parallel composition is to somehow argue about the security of multiple games that are played together simultaneously.

A typical use case of parallel composition of games is to show that *multi-instance* games reduce to the security of a single-instance game, as a consequence transforming multiple adversaries into a single adversary. may invoke the BDFKK Multi-instance Lemma [11].

3.5 Semantic security in SSP

Similar to IND-CPA security, we now introduce a modular definition of semantic security in SSP (Figure 3.5). One subtle but important difference is that we use a **Sim** package as our simulator, instead of an algorithm, i.e. \mathcal{S} in Chapter 2, Definition 2.4. This distinction is important because simulators are often stateful, and hence expressing them as packages is natural. If we use algorithms, as in the prior definition, we must instead pass state between them explicitly as inputs.

The rest of the packages used in our SSP-based definition of semantic security (Figure 3.4) have been defined previously in the games for IND-CPA. However, note that we use the **Zeroer** package here in a different context, to fulfill the role of the ideal functionality in semantic security, which extracts only the length of the message, and passes it to the simulator (in unary encoding). Also observe that the real semantic security game Gsem-sec^0 (on

the left of Figure 3.5) is equivalent to the real IND-CPA game Gind-cpa^0 . Therefore, in our equivalence proof (semantic security \iff IND-CPA) in Section 3.6, we restrict our attention to transforming only the ideal games of these notions.

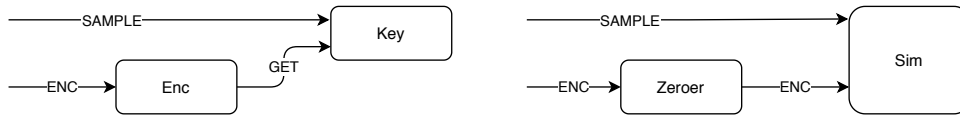


Figure 3.4: Definition of semantic security of probabilistic encryption se with the real game Gsem-sec^0 on the left and ideal game $\text{Gsem-sec}_{\text{Sim}}$ on the right.

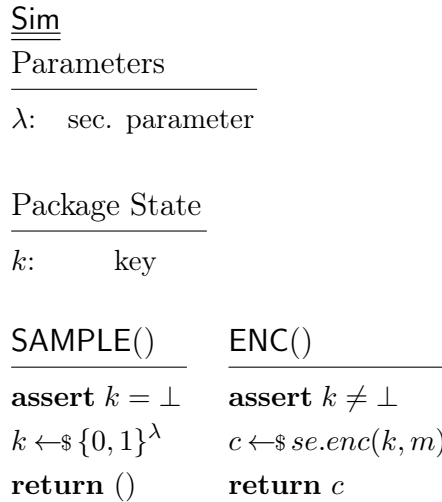


Figure 3.5: Oracles of simulation package Sim

Definition 3.2 (Semantic security). A probabilistic encryption scheme se is semantically secure if for all PPT adversaries \mathcal{A}

$$\text{Adv}(\mathcal{A} \rightarrow \text{Gsem-sec}^0, \mathcal{A} \rightarrow \text{Gsem-sec}_{\text{Sim}})$$

is negligible.

3.6 Proof example: Semantic security and IND-CPA equivalence

To prove that the notions IND-CPA and semantic security are equivalent, we must prove two implications, namely that IND-CPA security implies semantic security (Lemma 1), and semantic security implies IND-CPA security (Lemma 2).

Theorem 3.3 (IND-CPA security and semantic security equivalence). *A symmetric encryption scheme se is IND-CPA secure if and only if se is semantically secure.*

At a high level we first show that IND-CPA security implies semantic security with the construction of a concrete simulator package Sim . For the other direction, we show that assuming *some* simulator Sim satisfying the notion of semantic security, we can derive the IND-CPA notion.

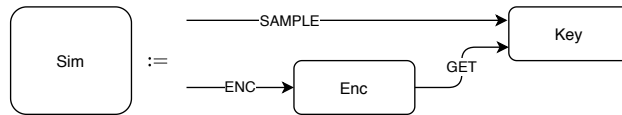


Figure 3.6: Definition of Sim package (construction) for Lemma 3.4

Lemma 3.4 (IND-CPA security implies semantic security). *A symmetric encryption scheme se is semantically secure if it is IND-CPA secure.*

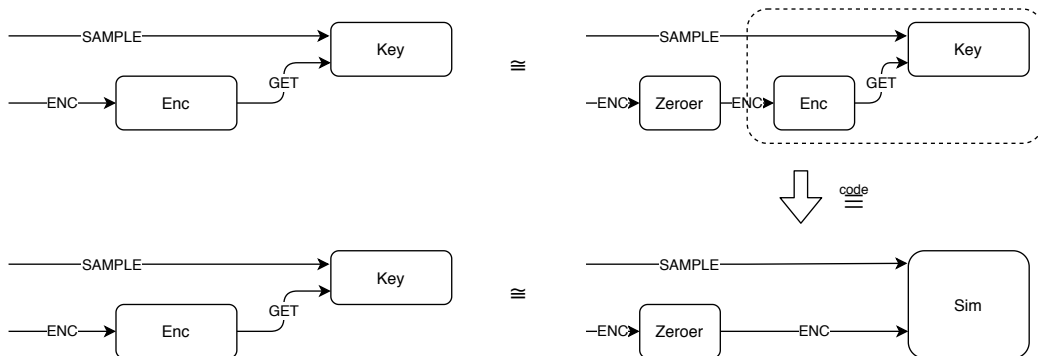


Figure 3.7: Proof overview of Lemma 3.4 (IND-CPA \Rightarrow semantic security)

Claim 3.5 (Ideal IND-CPA game and simulation game equivalence). Gind-cpa_{se}^1 is code equivalent to Gsim-sec_{se} . Note that in order to show this, we simply define Sim to be $(\text{Enc} \circ \text{Key})$ as in Figure 3.6. Since the simulator Sim is essentially defined as this subgraph (which, in fact turns out to be the real game), we say that the two games are *trivially* code equivalent.

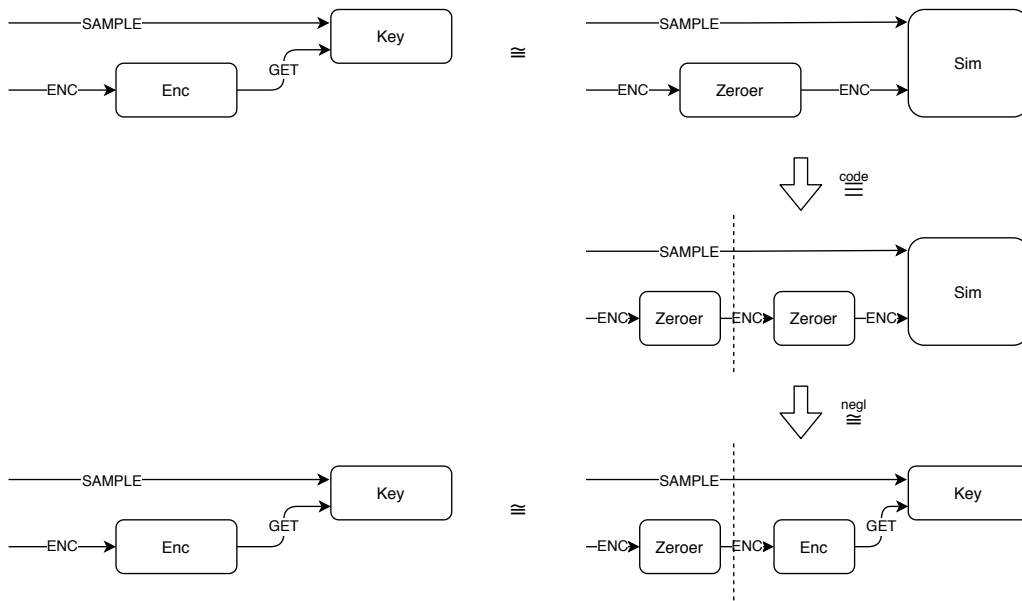


Figure 3.8: Proof overview of Lemma 3.6 (Semantic security \rightarrow IND-CPA)

Lemma 3.6 (Semantic security implies IND-CPA security). A symmetric encryption scheme se is IND-CPA secure if it is semantically secure.

This direction of the proof is somewhat more interesting - given some simulator Sim such that Gind-cpa^0 and Gind-sim are indistinguishable, we need to show that we can transform Gind-sim into the ideal IND-CPA game Gind-cpa^1 . Lemma 3.6 directly follows from Claim 3.7 and Claim 3.8 which we state now.

Claim 3.7 (Zeroer composed with simulation game equivalence). Notice that in order to argue the code equivalence of this hop, we only have to prove that $\text{Zeroer} \circ \text{Zeroer}$ is code equivalent to Zeroer .

Proof. To see that they are equivalent, we observe that Zeroer receives some message m and always produces $0^{|m|}$, independent of the contents of m . If we

do this several times by chaining multiple instances of `Zeroer` (by `ENC` calls) with `get` that the length of the messages are preserved and that the output is still $0^{|m|}$ (`Zeroer` is idempotent). \square

Claim 3.8 (Reduction to Real IND-CPA game). To the right of the dashed line is the assumption we reduce to, i.e. we reduce to our original assumption that `Gind-cpa0` and `Gind-sim` are indistinguishable.

3.7 State separating constructions (SSC)

While modular games are used in cryptographic proofs, they often mean that larger games are broken down to smaller *games*, without cryptographic *constructions* themselves being modularized into packages. Hence we refer to these as state separating constructions (SSCs), a notion introduced by Brzuska and Oechsner [12].

An advantage of splitting games into packages is that they allow us to reason about systems in a modular fashion. Instead of reasoning with a single, large monolithic system, we may now break them into smaller, reusable packages that communicate via oracle calls. SSCs allows us to analyze games by their components (expressed as packages) which is often simpler and much clearer. The use of constructions not only makes the structure of the protocol more apparent, but enables more syntactic reasoning and hence, reduces the number of proof steps.

3.8 Reductions in SSP

In SSP, reductions are thought of as *packages* that sit between an adversary and the assumption we reduce to. Let us take the example of the reduction step (Claim 3.8) to illustrate some of the properties of reductions in SSP. In the graphs of this reduction step, we observe that a dashed line was used to specify a *cut* in the call graph of the game `Gind-cpa0`. To the left of the dashed line is the actual reduction, and to the right is the *assumption* that we are reducing to. In this specific example, the reduction consists of a single package `Zeroer`, with the assumption being the indistinguishability of `Gind-cpa0` and `Gind-sim`.

Thinking about reductions this way allows to use part of a construction itself as the reduction. While the description of a state separating *construction* may be more detailed, we ultimately save cost when reasoning about reductions - since we may just point to a call graph and specify a cut, instead of creating a separate, new transformation. The other advantage, is

that reductions (and their corresponding assumptions that they reduce to) become much clearer in our understanding of the protocol's security, because we literally *see* them depicted in SSP call graphs.

3.9 Conventions and style

Naming

In this work we usually denote oracle names by upper case letters, e.g. **ENC**. On the other hand packages are capitalized e.g. **Enc**, **Zeroer**. Quite frequently we choose to omit the indices of instance packages from their names, for convenience and to avoid clutter in SSP call graphs. However, it is important to note that technically there cannot be multiple packages of a package **P**, but only multiple *instances* of **P** in a game definition. For instance, in Figure 3.8, notice that we used two **Zeroer** package instances, but chose to omit their indices (which are unique).

Adversaries

We tend to omit adversary packages from game definitions, since in the typical case we quantify over all efficient adversaries. Moreover, in games hops we quantify over different adversaries and hence should technically use a different adversary package there, which quickly becomes cumbersome. However, we have included adversary packages in Figure 3.1 for completeness, but will omit them in future definitions.

Chapter 4

Garbled Circuits

The idea of *garbling* or *encoding* a circuit was first introduced by Yao in 1986 in his seminal paper [35]. The security of Yao’s garbling scheme was proven only much more recently (in 2009) by Lindell and Pinkas [24], and has ever since had several other proofs. The concept of garbling itself has been described and formulated more abstractly, with increasing generality, as a primitive by Bellare, Hoang and Rogaway (BHR) [6], and later as randomized encodings, by Applebaum [2]. However, our focus in this work is Yao’s garbling scheme, and the purpose of this chapter is to introduce the scheme and build an understanding of how it is used for secure multi-party computation (MPC).

We begin by introducing the concept of secure multi-party computation (MPC) and relevant terminology (Section 4.1). Next, we describe Yao’s garbling scheme (Section 4.2), first by explaining the garbling of a single logical gate (Section 4.2.1), how we represent Boolean circuits (Section 4.2.2), and then circuit garbling (Section 4.2.3) and garbled circuit evaluation (Section 4.2.4). We then introduce the concept of *oblivious transfer* (OT) (Section 4.3), together with an example OT construction called Simple OT (Section 4.3.1), which can be combined with Yao’s garbling scheme to obtain MPC (which we describe in Section 4.4). Lastly, we discuss garbled circuits optimizations and their security assumptions to better understand the impact of optimizations on the respective security proofs (Section 4.5).

4.1 Secure multi-party computation

Secure multi-party computation (MPC) is an area of cryptography which develops techniques that enable multiple mutually distrusting parties to jointly compute some function without revealing their respective inputs. In Chap-

ter 2, we have already seen an example of an MPC protocol, which securely computed the AND of three parties' input bits.

Recall that participants in MPC protocols may potentially be adversaries. We think of adversaries that try to break the system by only observing the execution of the protocol (the view), as honest but curious adversaries. This is also known as the *semi honest* setting, whereby adversaries still comply to the protocol. In this work we restrict our focus to protocols that are secure against passive adversaries because they are the starting point for understanding MPC constructions, and are simpler than actively secure protocols, while being sufficiently interesting. Moreover, passively secure MPC protocols can be generically transformed into an enhanced protocol that is secure against malicious adversaries [8], although this may not be efficient in practice.

When only 2 parties engage in MPC, we refer to this as 2-party computation (2PC). In the following section (Section 4.2) we begin our discussion of Yao's garbling scheme, which is a useful building block for 2PC.

4.2 Yao's garbling scheme

The correctness of Yao's garbling scheme states that given a Boolean circuit C and its garbling \tilde{C} , the evaluation of C on an input bits x yields the same output as evaluating x on the garbled circuit, i.e. for all $x, C(x) = \tilde{C}(x)$. The goal of *garbling* is to in some way *encode* a Boolean circuit, for it to be evaluated later on encoded inputs from both parties. We can think of Yao's garbling scheme as being used by 2 parties namely a *garbler* G who garbles the circuit and the inputs of both parties, and an *evaluator* E who evaluates the garbled circuit on their joint garbled inputs. The security goal of a garbling scheme (informally) is for an evaluator E evaluating the circuit not to learn the inputs of G , and conversely G should not learn anything about the inputs of E . Moreover, E should not learn any information about the intermediate values of the computation while evaluating a garbled circuit, apart from what can be deduced from the output.

At this point, an observant reader might raise the question, how does the garbler G even garble their joint inputs if it is *not* supposed to know E 's inputs? Roughly speaking, G overcompensates by garbling all possible inputs and allows E to request for its garbled inputs via a protocol known as *oblivious transfer*. However, for the time being we kindly ask the reader to take on faith that this "blind" transfer of wirekeys is possible, and this protocol will be covered at a later point (Section 4.3).

To understand the concept of garbling, first we describe the process of garbling a single logic gate in the following section, since gates are the fun-

damental components of Boolean circuits.

4.2.1 Gate garbling

We depict the process of gate garbling in Section 4.1, whereby we use the example of garbling an AND gate (note that we could have used any logical operation as an example). Furthermore, we denote the left and right input bits of the gate, by z_ℓ and z_r respectively.

z_ℓ	z_r	$z_j = z_\ell \wedge z_r$	Z_ℓ	Z_r	Z_j	garbled
0	0	0	$Z_\ell(0)$	$Z_r(0)$	$Z_j(0)$	$E_{Z_r(0)}(E_{Z_\ell(0)}(Z_j(0)))$
0	1	0	$Z_\ell(0)$	$Z_r(1)$	$Z_j(0)$	$E_{Z_r(0)}(E_{Z_\ell(1)}(Z_j(0)))$
1	0	0	$Z_\ell(1)$	$Z_r(0)$	$Z_j(0)$	$E_{Z_r(1)}(E_{Z_\ell(0)}(Z_j(0)))$
1	1	1	$Z_\ell(1)$	$Z_r(1)$	$Z_j(1)$	$E_{Z_r(1)}(E_{Z_\ell(1)}(Z_j(1)))$

Figure 4.1: Truth table of AND gate (left) and its garbling (right)

To garble a gate, a garbler G first hides the semantic value (or bit) of each input wire with a *wirekey* (sometimes referred to as a label, and is typically a symmetric key), where each wirekey is sampled independently and uniformly at random. We think of wirekeys as produced by maps of the form $Z : \{0, 1\} \rightarrow \{0, 1\}^\lambda$ (given a bit b , $Z(b)$ produces the corresponding wirekey, a bitstring of length λ). All that the garbler has done up to this point is encode each bit value with a wirekey. To actually garble the gate, G encrypts each output wirekey with both of its corresponding input wirekeys. Concretely, to garble the first row of the gate in Figure 4.1, the garbler starts by encrypting the output wirekey $Z_j(0)$ with Z_ℓ , and the resulting ciphertext is then encrypted with $Z_r(0)$, i.e. $E_{Z_r(0)}(E_{Z_\ell(0)}(Z_j(0)))$. This procedure is referred to as *double encryption*. The column **garbled** (highlighted in gray) is what we refer to as a *garbled gate*, but with the additional step that these rows are randomly permuted, since otherwise an adversary could trivially know which pair of inputs corresponds to which ciphertext/row.

To see how a garbled gate is evaluated, let us suppose that an evaluator E receives the garbled gate **garbled** and a wirekey corresponding to its input bit 0, i.e. it obtains as *garbled input* $Z_\ell(0)$). Additionally, E receives the garbler’s garbled input, lets say $Z_r(1)$ (note that what E sees here is a wirekey value of λ random bits), corresponding to the garbler’s input of 1. With wirekeys $Z_\ell(0)$ and $Z_r(1)$, E is only able to decrypt or “unlock” the third row or ciphertext, although in practice the rows would be in random order, so E tries to decrypt each row until it is successful (implicit in this definition is that the underlying encryption scheme should have the property of *decryptability*, i.e.

upon decryption the decryptor knows if it was successful or not¹). Intuitively, notice that the security of a garbled gate relies on an adversary knowing only one of the wirekeys, since if it knew both, it could learn the output. For this reason, the example we used ensured that each party received only one wirekey corresponding to either the left or right input wire of the gate.

Before discussing the process of circuit garbling, we briefly explain how we choose to represent Boolean circuits.

4.2.2 Circuit representation

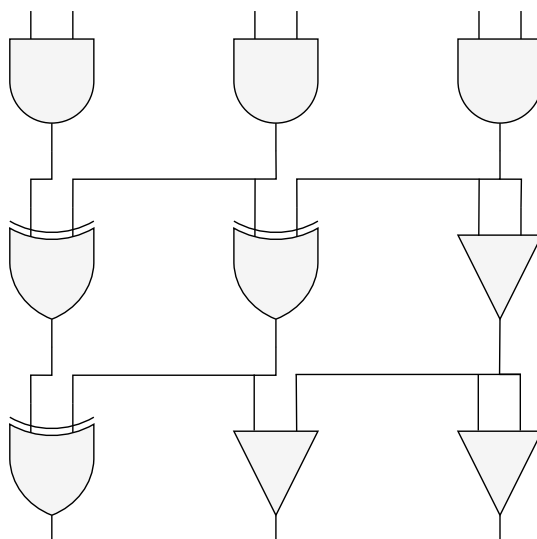


Figure 4.2: Boolean circuit C

A Boolean circuit can be thought of as a Directed Acyclic Graph (DAG), i.e. a graph with no loops, with each node representing a unit of computation performing a specific operation op (e.g. AND/XOR). Moreover, we fix all gates to have 2 input wires, a left and a right wire which we denote by ℓ and r , that functions, which given a wire returns its bit value 0 or 1. We denote the depth of a circuit by d , and its width by n . While there are many ways to represent them, a convenient way to think of them is as a $d \times n$ matrix, i.e. each layer $i \in 1..d$ of a circuit has a fixed width n , with each entry being a gate. We find this representation convenient in defining the composition of layers (to obtain a circuit), and we will see how this lends itself to making the

¹One way to achieve this is to append 0^λ to the wirekeys, and upon successful decryption λ many zeroes appear. However, as we will see with the *point and permute* optimization, this may be avoided altogether.

security proof more elegant. As a running example for the following sections, we use the 3×3 Boolean circuit depicted in Figure 4.2, whereby the first layer consists of AND gates, the second layer with 2 XOR gates and a single identity gate (which simply forwards its left input).

4.2.3 Circuit garbling

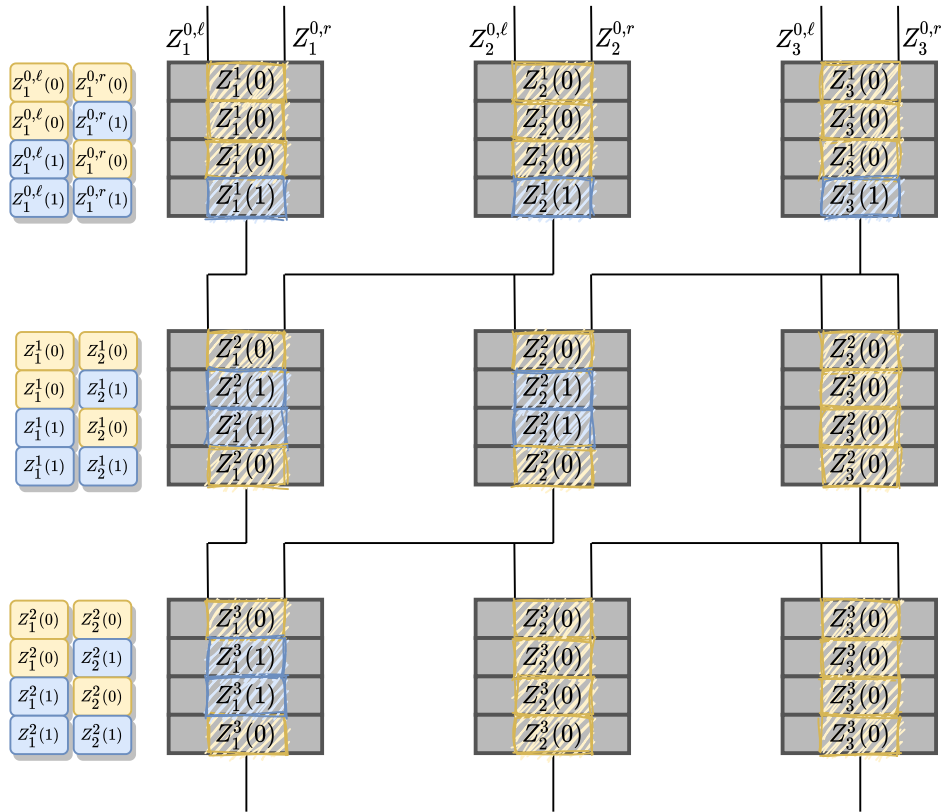


Figure 4.3: Garbled circuit \tilde{C}

We describe the process of circuit garbling with the aid of Figure 4.3, which depicts the garbling of the original Boolean circuit C (Figure 4.2). Note that for convenience we have *not* shuffled the ciphertexts of each garbled gate. At a high level, to garble a circuit, the procedure of gate garbling performed for each gate in topological order, beginning with input gates of the first layer (which do not rely on their input coming from another gate), and then garbling the next layer based on the outputs of the previous layer, iteratively till the last layer of the circuit.

In Figure 4.3, we have illustrated the input wirekeys of gates on the left of the first column of gates, with yellow wirekeys indicating a bit semantic of 0 and blue wirekeys indicating a bit semantic of 1. The rows of the garbled circuit are depicted with diagonal lines going through them to indicate that what is visualized is actually the *ciphertexts* that contain output wirekeys (i.e. double encrypted wirekeys), and not the output wirekeys themselves.

Initially, the garbler G generates pairs of wirekeys for the input layer (since our representation fixes gates to have 2 inputs each, G would generate $2 \times 2 \times n = 4n$ wirekeys). In the diagram we have denoted these wirekeys as outputs of Z_j^0 for $j \in 1 \dots n$ (with $n = 3$), that come from a “zeroth” layer. G also generates output wirekeys for the rest of the gate, which it double encrypts using the input wirekeys that are output wirekeys of previous layers. Notice that in Yao’s garbling scheme, these wirekeys are the only state that circuit layers share. Hence the output wirekeys of the previous layer must be generated before garbling subsequent layers, and garbling proceeds layer by layer.

4.2.4 Garbled circuit evaluation

In order to evaluate a circuit, an evaluator E has to first obtain all input wirekeys necessary for the computation, i.e. the garbler’s garbled inputs as well as its own garbled inputs. It then “unlocks” each gate in the first layer with the wirekeys, obtaining the output wirekeys that can be in turn used to decrypt gates of the next layer. In Figure 4.4 we see this process of “unlocking” a row in the garbled gate by highlighting the unlocked ciphertext in green.

Note that for our example on garbled circuit evaluation (Figure 4.4), we have evaluated the garbled circuit in Figure 4.3 with the encoded inputs of the bits $(1, 0)$, $(1, 1)$ and $(0, 1)$, for each gate of the first layer respectively. Note that although we have highlighted the wirekeys according to their bit semantics (yellow for 0 and blue for 1), the evaluator E does not know this information (i.e. the bit semantics of the wirekeys), except for its own garbled inputs, the intermediate wirekeys it unlocks and the final result of the computation. The keys that E is able to unlock are known as *active* keys, and the wirekeys it cannot reach are called inactive. We say that the evaluator knows only *active semantics* instead of bit semantics.

Note that the garbler provides the evaluator E with a decoding table (on the bottom left of Figure 4.4) which maps output wirekeys to their bit values, so that the evaluator can derive the output values.

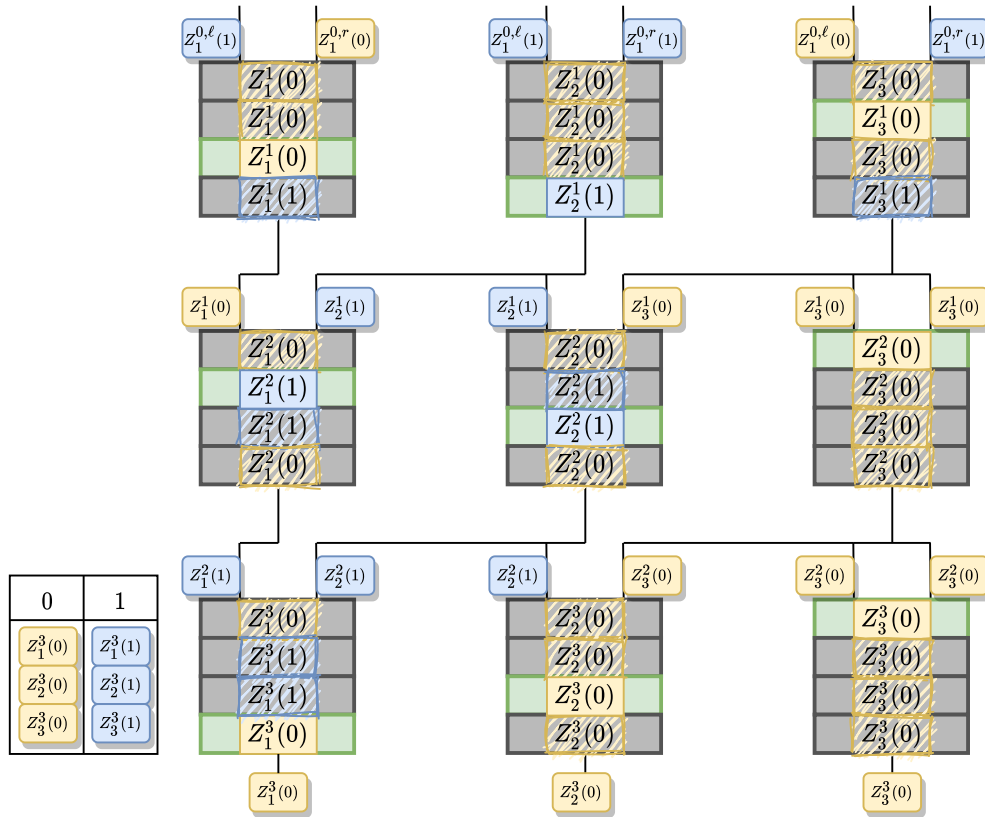


Figure 4.4: Evaluation of the garbled circuit on inputs $(1, 0)$, $(1, 1)$ and $(0, 1)$

4.3 Oblivious Transfer

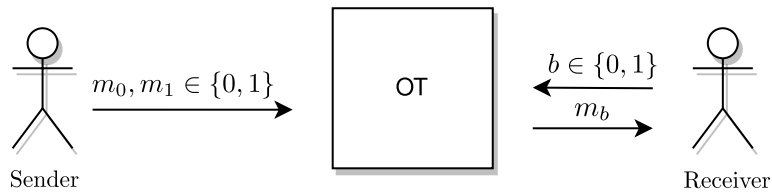


Figure 4.5: Oblivious Transfer

Oblivious transfer (OT) is a protocol involving 2 parties, namely a sender and receiver, whereby the sender transfers one of two messages m_0, m_1 to a receiver, based on the receiver's selection bit b (depicted in Figure 4.5). The

goal of OT is to ensure that the sender learns nothing about b and the receiver learns nothing about m_{1-b} .

4.3.1 Simple OT

Simple OT (Figure 4.7) is an efficient 1-out-of-2 OT protocol due to Chou and Orlandi [14], and partly resembles the Diffie-Hellman key exchange protocol (shown in Figure 4.6).

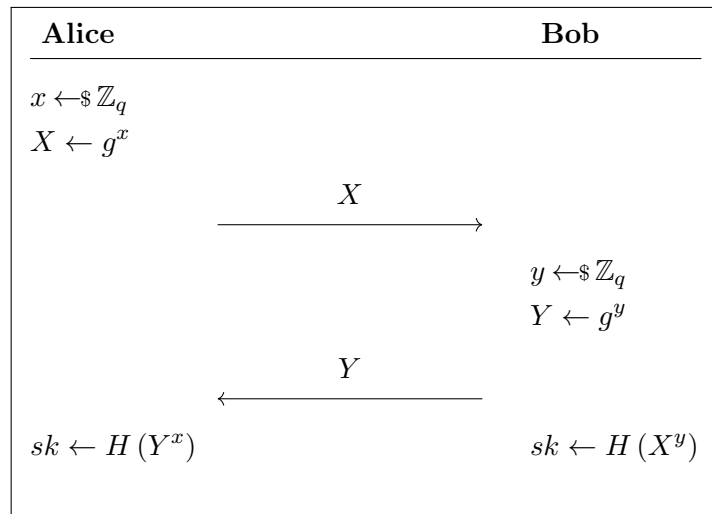


Figure 4.6: Diffie-Hellman key exchange protocol

In contrast to the standard Diffie Hellman key-exchange protocol, the Simple OT (Figure 4.7) protocol adds a few additional steps for computing the keys k_0, k_1 , for the purpose of encrypting messages m_0 and m_1 , respectively. The idea is that the receiver is able to compute only one of these keys, namely the key based on its choice b , only with the information that it has obtained through the interaction. The way the first key k_0 is computed is identical to key derivation in the Diffie Hellman protocol. The hash function H converts the derived secret g^{xy} into a symmetric key sk to be used for encryption. Note that (enc, dec) is a symmetric key authenticated encryption (AE) scheme.

4.4 MPC from garbled circuits

Figure 4.8 illustrates Yao's protocol for MPC. The evaluator E holds m inputs, whereas the garbler G has the remaining $n - m$ inputs. In (b), G

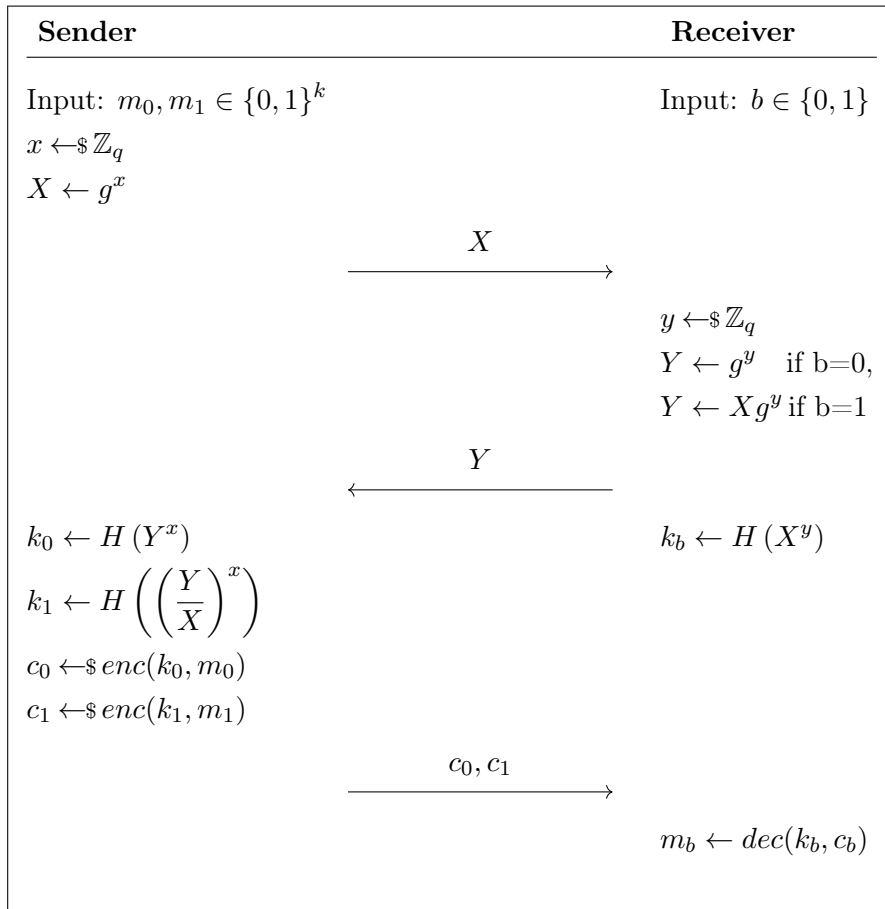


Figure 4.7: Simple OT protocol

generates wirekeys for the input layer. The algorithm gb garbles a circuit, whereas the algorithm gev is used to evaluate a the garbled circuit on the joint garbled inputs $\tilde{x} := x[0], x[1], \dots, x[n]$ (we use only notation here and have not provided a formal definition for gb and gev).

Yao’s protocol for MPC may be summarized by the following steps:-

1. The function f (that is to be computed) is first represented as a circuit C .
2. The garbler G garbles C and sends to the evaluator E the garbled circuit \tilde{C} and its garbled inputs ((d) of Figure 4.8).
3. E then requests from G to receive its garbled inputs via OT.
4. E evaluates the garbled circuit on the joint garbled inputs $\tilde{x} := x[0], x[1], \dots, x[n]$, to obtain the garbled output \tilde{y} , which it decodes into y , and shares the

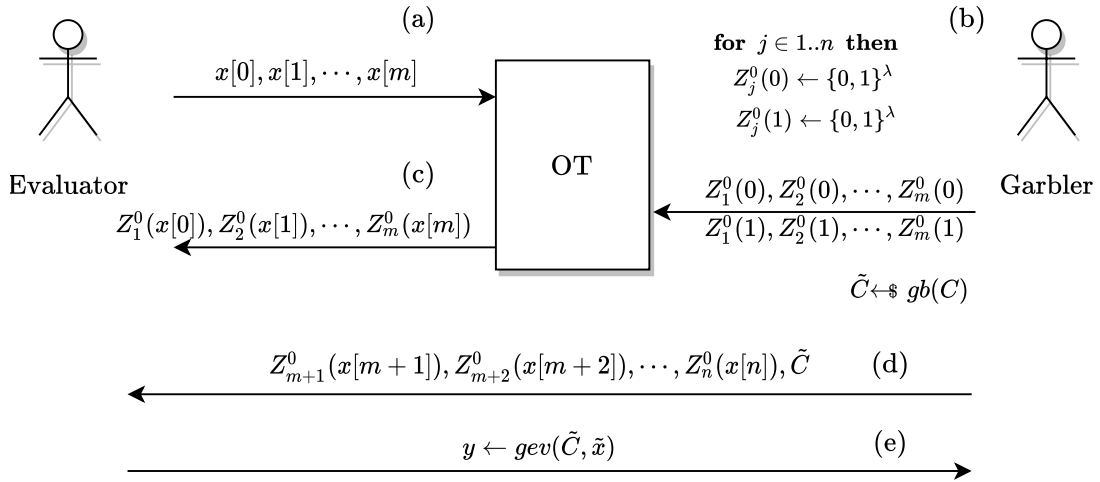


Figure 4.8: Yao's protocol

result with G .

4.5 Garbled circuits optimizations

Garbling schemes have had numerous optimizations and are widely used in practice, and hence must be considered especially with regard to security. These optimizations tend to make additional assumptions which may affect the security proof of garbling schemes that use them. In particular we present 3 garbled circuits optimizations that are used in practice, and discuss their impact on proofs of security.

Point and permute

The point and permute optimization by Beaver et al. [5] removes the need to decrypt every row of a garbled gate. The way this is achieved is by associating a permutation bit to the wirekeys to establish an ordering of the ciphertexts, meaning that they are ordered according to the permutation bits instead of a random shuffle. In practice, the permutation bit is tacked on the least significant bit (LSB) of wirekeys, so that they do not need to be transferred (via OT) separately. An evaluator reading of the permutation bits from the wirekeys it has obtained simply looks up that index in the garbled gate to find the ciphertext it needs to decrypt. The beauty of this optimization is that it has no computational assumptions, and hence can be used with most

other optimizations, or applied to any garbling scheme without affecting the scheme’s security proof in any significant way.

Garbled row reduction

There are namely 2 garbled row reductions techniques (that work for any gate operation) - one that reduces the size of a garbled gate to 3 ciphertexts called GRR3 [30], and another called GRR2 [31] which reduces the size further to 2 ciphertexts. The trick employed by GRR3 is to always set the first ciphertext of a garbled gate to evaluate to the all zeroes, which omits the need to transmit it. This involves setting the first output wirekey Z_j (not necessarily the row corresponding to the input $(0, 0)$) to be $enc_{Z_\ell}^{-1}(enc_{Z_r}^{-1}(0^\lambda))$. This optimization is fine since it assumes that the encryption algorithm used has an inverse given a fixed key, which is simply the decryption function. On the other hand GRR2 is not adaptively secure [20] (but since the focus of this work is on passively secure schemes, we do not discuss this further).

FreeXOR and Half gates

The FreeXOR optimization introduced by Kolesnikov and Schneider [22] avoids garbling XOR gates altogether, making their evaluation free of cost. This optimization is particularly useful for circuits that consist mostly of XOR gates, such as the AES circuit, in which the fraction of XOR gates in the circuit is approximately 82%².

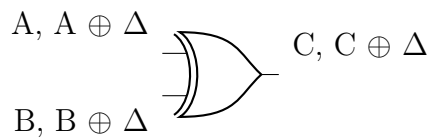


Figure 4.9: XOR gate garbling with FreeXOR

FreeXOR involves picking a global wirekey offset Δ that is XORed to all wirekeys. The output wirekeys of XOR gates are then assigned such that the wirekeys preserve XOR semantics, while other gates are garbled with GRR3 (which happens to be compatible with FreeXOR). Concretely the output wirekey C ’s zero-wirekey value is $C = A \oplus B = (A \oplus \Delta) \oplus (B \oplus \Delta)$, and its one-wirekey is $C \oplus \Delta = A \oplus (B \oplus \Delta) = (A \oplus \Delta) \oplus B$. However, the use of the same offset Δ raises an issue of security, since garbling such a gate will

²This statistic was obtained by counting the number of XOR gates of the AES circuit provided by the website Bristol Fashion MPC Circuits [1]

involve encrypting an output wirekey with input wirekeys that are related by Δ . To fix this, the authors introduce an additional assumption of correlation robust hash functions, whereby output wirekeys of the circuit (final layer) are hashed to “destroy” the offset. Moreover, notice that circuit layers no longer only share wirekeys, they now must also share the offset Δ , which also affects the security proof of a garbling scheme that uses FreeXOR. For these reasons, the circuit layers cannot be proven individually and independently (via individual layer security games) and hence cannot be composed in parallel. Thus, a security proof would involve a reduction over the entire circuit (security game).

Subsequent optimizations such as FlexOR [21] and Half gates [36] optimize garbled circuits further by ensuring that the size of all garbled gates are either 0, 1, or 2 ciphertexts. However, they build upon FreeXOR and hence requires the same correlation robust hash function assumption.

Chapter 5

Security of Garbled Circuits

We mentioned (in Chapter 4) that correctness of garbled circuits states that given a circuit C and its garbling \tilde{C} , for all inputs $x, C(x) = \tilde{C}(x)$. To complete this definition we follow Bellare, Hoang & Rogaway (BHR [6]) who introduced the syntax for garbling schemes (Section 5.1). We follow the BHR syntax, but use Brzuska's & Oechsner's approach [12] and formulate syntax in the language of packages (Section 5.2).

5.1 BHR syntax

The BHR syntax for garbling schemes allows us to describe garbling schemes within an abstract framework, giving us a language to precisely specify the correctness and security properties of garbling schemes.

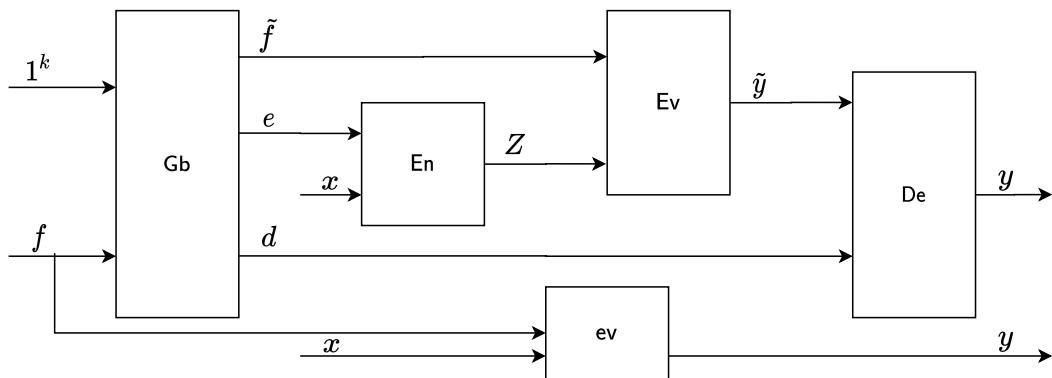


Figure 5.1: BHR syntax (Adapted from [6])

Definition 5.1 (BHR Garbling Schemes Syntax). A garbling scheme is a five-tuple of polynomial time algorithms $(\text{Gb}, \text{En}, \text{De}, \text{Ev}, \text{ev})$, such that:

- Gb (Garbling algorithm) is a probabilistic algorithm that takes as input a security parameter 1^k and function f to produce \tilde{f}, e, d .
- En (Encoder) takes as input string e which describes encoding information, the input z , and outputs the resulting garbled input Z .
- Ev (Evaluator) takes as input an encoded input Z and a garbled function F and outputs a garbled output $\text{Ev}(\tilde{f}, Z) = \tilde{y}$.
- De (Decoder) takes as input string d which describes decoding information, a garbled output \tilde{y} and outputs the final output y .
- ev (Canonical evaluation function) takes as input the (un-garbled) function f and input x and outputs y .

The correctness of a garbling scheme states that for all inputs z , the canonical evaluation function $\text{ev}(f, z)$ (which takes the un-garbled function and input) and the composition of the garbling algorithms, output the same value. Figure 5.1 depicts the BHR syntax visually as interactions between algorithms of a garbling scheme (represented by boxes).

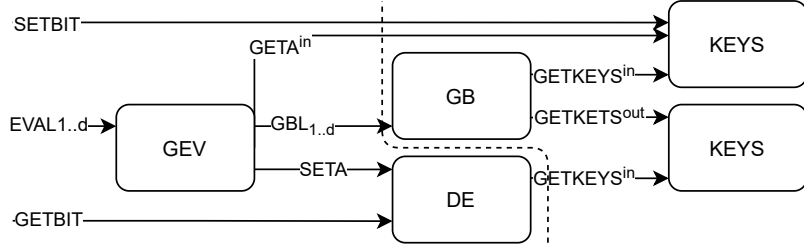
Out of the possible security properties a garbling scheme may guarantee (i.e. privacy, obliviousness and authenticity), we focus on *privacy*, which is the basic security goal of garbling schemes. Privacy states that a party who obtains the garbled function \tilde{f} , garbled input Z and the decoder d , only learns the output of the function y and not anything else, except for some “allowed” side information Φ , which in our case is $\Phi_{\text{circ}}(f) = f$, since we assume the circuit to be public.

5.2 Correctness of garbling schemes

To formulate the BHR syntax within SSP we begin by (roughly speaking) converting the tuple of algorithms $(\text{ev}, \text{Ev}, \text{Gb}, \text{De}, \text{En})$ into packages. Notice that the algorithms share state among each other (notice the arrows between algorithms in Figure 5.1) in order to function as a garbling scheme. Hence, we find it natural to represent them as an SSP game with packages corresponding to algorithms.

Definition 5.2 (Garbling scheme syntax in packages). A garbling scheme consists of packages GB , such that GEV if the game in Figure 5.2 is statistically indistinguishable from the evaluation of the ideal correctness game F_d^{gbl} (Figure 3 of [12])

We describe the correctness of a garbling scheme with reference to Figure 5.2. We allow an adversary to first set its inputs via SETBIT before issuing


 Figure 5.2: Real correctness game $\text{GCORR}(\text{GB}, \text{GEV})$ (Adapted from [12])

$\frac{\text{DE}}{\text{SETA}(j, k)}{\text{assert } k_j = \perp}$ $k_j \leftarrow k$ $\text{return } ()$	$\frac{\text{KEYS}}{\text{GETBIT}(j)}{\text{assert } z_j \neq \perp}$ $\text{return } z_j$	$\frac{\text{GETINA}^{\text{in}}(j)}{\text{assert } z_j \neq \perp}$ $\text{assert aflag}_j = 1$ $\text{assert } Z_j \neq \perp$ $\text{return } Z_j(1-z_j)$
$\frac{\text{GETBIT}(j)}{\text{assert } k_j \neq \perp}$ $Z \leftarrow \text{GETKEYS}(j)$ $\text{if } Z(0) = k_j :$ $\quad \text{return } 0$ $\text{if } Z(1) = k_j :$ $\quad \text{return } 1$ $\text{return } ()$	$\frac{\text{GETA}^{\text{out}}(j)}{\text{assert } z_j \neq \perp}$ $\text{aflag}_j \leftarrow 1$ $\text{if } Z_j = \perp \text{ then}$ $\quad Z_j(0) \leftarrow_{\$} \{0, 1\}^\lambda$ $\quad Z_j(1) \leftarrow_{\$} \{0, 1\}^\lambda$ $\text{return } Z_j(z_j)$	$\frac{\text{GETKEYS}^{\text{in}}(j)}{\text{assert } z_j \neq \perp}$ $\text{return } z_j$ $\frac{\text{GETKEYS}^{\text{out}}(j, z)}{\text{assert } z_j = \perp}$ $z_j \leftarrow z$ $\text{return } ()$

Figure 5.3: Oracles of the DE package (left) and KEYS package (right)

an $\text{EVAL}_{1..d}$ to the package GEV (which represents the evaluation algorithm, which we abuse by also referring to it as an *evaluator* in this section), and a GETBIT oracle, in order obtain the result of the garbled circuit evaluation. Note that SETBIT and GETBIT are both directly implemented in KEYS which is in charge of sampling wirekeys, hence it makes sense to include input encoding in the same package (we don't garble inputs after garbling a circuit - that would bring us into the land of actively secure garbling schemes, beyond the scope of this work).

Recall that an evaluator only knows active semantics and not bit se-

mantics (unlike the garbler). For this reason we provide **GEV** access to a GETA^{out} oracle in order to obtain the active keys (the name **GETA** for “get active”). Notice that the **KEYS** package provides **GETKEYS** oracles which sample wirekeys when they are called the first time. Hence, GETA^{out} also samples the wirekeys that will be used in garbling before returning the active wirekey to the evaluator. The evaluator is given an oracle **GBL** to garble a circuit (more accurately, it performs d **GBL** queries for each layer of a circuit, but we won’t speak of circuit layers just yet). The decoder package **DE** comes into play when the evaluator could set the active key (via **SETA**) of a gate (inputs and gates are indexed by the variable j). Notice that if it setting the active key of a gate affects the outcome **GETBIT**, namely, it would return void () if *none* of the wirekeys used in garbling were set as the active key, meaning that it makes sense for the evaluator to only set active keys according to what it has already obtained by unlocking a gate.

Notice that in most cases the order in which calls are made to oracles matter. In order to do this we asserts have been place at the beginning of the oracles to check for certain conditions. The interested reader may verify that changing the order of calls in certain cases (e.g. **GBL** before **GETA**) does not change the behaviour of the game. We now turn to security definitions for garbling schemes.

5.3 Security of garbling schemes

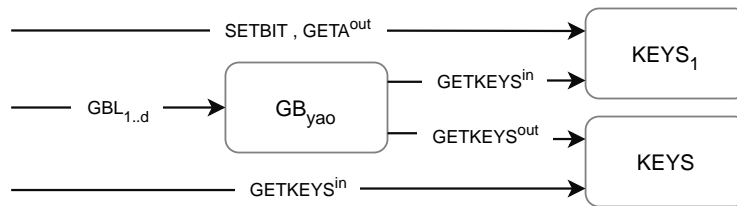


Figure 5.4: $\text{SEC}^0(\text{GB})$

Notice in our definition of correctness (Figure 5.2) we included a dashed line to denote a cut, which specifies exactly the real security game (Figure 5.4). Intuitively, what this intends to say that we may consider an evaluator as an adversary.

Definition 5.3 (Garbling scheme security). A garbling scheme $gs = (\text{GEV}, \text{GB})$ is secure if there exists a PPT simulator **SIM** such that for all PPT adversaries

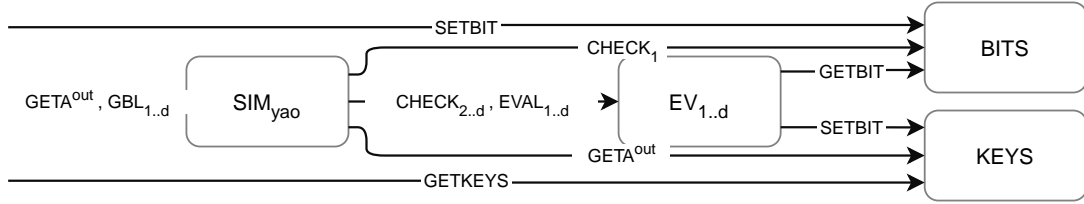


Figure 5.5: $SEC^1(SIM)$

\mathcal{A} ,

$$\text{Adv}(\mathcal{A}; \text{SEC}^0(\text{GB}), \text{SEC}^1(\text{SIM}))$$

is negligible.

Theorem 5.4 (Yao security). *For all PPT adversaries \mathcal{A} , Yao’s garbling scheme is secure if*

$$\begin{aligned} \text{Adv}(\mathcal{A}; \text{SEC}^0(\text{GB}_{yao}), \text{SEC}^1(\text{SIM}_{yao})) &\leq \\ n \cdot d \cdot \text{Adv}(\mathcal{A} \rightarrow \mathcal{R}; \text{IND-CPA}^0(se), \text{IND-CPA}^1(se)) \end{aligned}$$

At a high level, the theorem states that security of Yao’s garbling scheme essentially rests on the security of a single gate, i.e. double encryption, which in turn relies on the IND-CPA security of the underlying encryption scheme it uses. In order to reduce to IND-CPA modularly we argue that circuit security reduces to layer security, since a circuit is composed of layers. For the security proof of Yao’s garbling scheme, the reader is kindly referred to view this proof in the SSP proof viewer (<https://cryptozoo.herokuapp.com/yao.html>)¹.

¹For future references the SSP proof viewer will most probably live in this repository: <https://github.com/kirthivaasan/ssp-proofviewer>

Chapter 6

Proof verification and understanding

Proofs are a fundamental way of communicating about truth with others. This notion considers proofs as a *social means* for understanding truth, by that we mean that proofs are written and understood by people, rather than machines.

In this chapter we discuss the idea of proof verification, specifically how cryptographic proofs can be verified by a machine, but how this is different from proof understanding by a human. We then present what we believe is a good synergy between humans and machines for better proof understanding. We argue that verifying a correct proof alone does not bring insight to *why* a statement is true. Hence, we draw a distinction between proof verification and proof *understanding*.

The chapter is organized as follows. We begin by giving an overview of cryptographic proof verification (Section 6.1). Within this section we also provide a discussion on how machines verify cryptographic proofs (Section 6.1.1) and their limitations (Section 6.1.2). We then go on to discuss proof communication (Section 6.2) and concepts such as information hiding (Section 6.3), proof structure (Section 6.4) and intent (Section 6.5), to increase the effectiveness of proof communication. Finally, we present a brief discussion on related work in the area of mathematical practice philosophy, pertaining to proof understanding and communication (Section 6.6).

6.1 Cryptographic proof verification

Recall that the goal of a cryptographic proof is to show that a real system (which uses the real cryptographic construction) is indistinguishable from an

ideal unbreakable system. Therefore, a central part of cryptographic proofs of security is a certain type of reasoning about the indistinguishability of systems, regardless of which framework is used. The steps of a proof are broken down into proofs of smaller statements (e.g. lemmas and claims) that employ reductions or code equivalences. The goal of verification is to check whether every step is logically sound, and whether all of the steps lead to showing that the original statement is true. These steps can often be difficult to follow when games are complex (and treated monolithically). The reader could be lost in trying to verify a proof when there are too many steps that are boring (e.g. the proof of CMC mode of operation [17]¹).

Therefore, we may confidently say that for the human reader, proof verification alone is not a very fun process. As stated by Halevi [17], we would like to instead transfer the effort of rote proof verification to machines. In the next section (Section 6.1.1), we discuss computer-aided proof verification in cryptography.

6.1.1 Formal methods and proof verification

There exist several important tools for cryptographic protocol verification using formal methods. These tools may be roughly divided into two categories, symbolic verification (e.g. Tamarin [29], ProVerif [9]), and computational verification (e.g. EasyCrypt [4]). Of these tools, we focus on computational security, which considers computational effects (via oracles) of a scheme/protocol and adversaries.

EasyCrypt is an interactive proof assistant for code-based game proofs. The tool works based on a formal logic (Probabilistic Hoare Logic) for reasoning about the indistinguishability of probabilistic programs (games). EasyCrypt *modules* are similar to SSP packages, in the sense that they store state, and all oracles within a module can access a shared state. Since, SSPs are a special case of the code-based games approach and use similar concepts with tools like EasyCrypt we have evidence to believe that they not only do well in proof communication, but are suitable for automated proof verification as well.

Moreover, there are a plethora of tools dedicated to proving/verifying the correctness of program implementations by checking if program specifications are functionally equivalent to their implementations (some examples of such languages/tools are F* and Coq). We believe that *code equivalence* steps in SSPs may be automated using such tools from the formal methods and

¹The author explicitly states that he does not expect any *sane* reader to verify the proof.

programming languages community.

6.1.2 Limitations of machine proof verification

What a (basic) machine proof checker may struggle with is in deciding which parts of a proof are more important and which are less important. Moreover, it may not be robust to changes in the proof, or see where the change is coming from, and what motivated it. This is of course unavoidable because the machine checker’s sole purpose is to check whether a proof is formally true, and if the steps taken in the proof are correct and derive the intended result.

While having a machine checked proof is excellent for ensuring the security proof is indeed correct and that there are no typos, it lacks in its ability to communicate insight into *why* the proof is correct. The goal of (machine) automated proof verification, should be (in our opinion) to sketch or “fill in” parts of the proof that are boring. We hold an optimistic view that machines not only can serve as good assistants but may further enrich our understanding of proofs and provide interesting results and insights ². At present, we may strike a good synergy with machines by giving them the tedious (and boring) parts of the proof to verify, while focusing on the more interesting parts, providing insights into a proof where a machine may miss/fail to do so.

6.2 Proof communication

Proofs tend to assume a lot of context - in fact there may never be a truly “self contained” proof ³ - since this would take a lot of time and information to convey and understand [16]. In order to make proof communication effective in cryptography, we present the following ideas, namely *information hiding*, *proof structure* and *intent*.

6.3 Information hiding

We believe that readers make multiple scans when attempting to understand a proof. At each iteration, the focus of the reader probably changes. Moreover on each scan the reader appreciates more detail. Hence the idea of

²With the advent of Artificial Intelligence (AI), we will probably have very many interesting developments in the field of automatic theorem proving. Also, it is probably wise to be polite to our (potential) future overlords.

³It takes roughly *20,000* steps to prove $2 + 2 = 4$ in ZFC set theory[15]

information hiding in a proof is crucial for effective communication. Seldom is it meaningful to organize all steps in a strictly linear fashion, with all details visible at each step. In order to achieve this traditionally, we may place the details such as definitions in different pages of a paper that presents the proof. In our design of the proof viewer we attempt to make navigating to find details of a proof more natural and convenient, in addition to allowing the user to hide information and details (Section 7.1).

6.4 Proof structure

The claims that a proof makes, along with their dependencies are all equally important (but perhaps not all equally *interesting*) in supporting a mathematical proof (unlike many other fields, mathematical arguments are strongly chained together, whereby a single missing link can destroy the validity of an argument). Therefore, it is useful to see the complete structure of a proof, and for the author to highlight parts of the proof that are considered important or interesting. The gist of a proof's structure probably becomes more apparent to a reader after multiple scans of a proof. We believe, that the process of understanding proof structure can be made more efficient (or even better avoided) if the structure was instead presented explicitly and visually. To achieve this, we may employ the use of *proof trees* which summarize the statements (e.g. lemmas and claims) of a proof concisely.

Moreover SSPs and SSCs are inherently visual objects and hence game hops have structural elements that are visual. We believe that clearer game structure entails clearer proof structure. Hence it is beneficial to use constructions (SSCs) where possible, since modular constructions make the structure of *games* more apparent. Moreover, (as discussed in Section 3.7) this reduces the overhead of steps that are reader must check, and also prefers syntactical steps, which are significantly easier to verify than non-syntactical steps, i.e. elaborate code transformations.

With regard to SSPs, we find that, SSP graph layouts are designed to be self-similar / symmetrical and hence possess structure as well, which is aided by *intent* (the topic of our next section).

6.5 Intent

A supporting concept of proof structure is *intent*. The function of intent is to provide a reader with the *raison d'être* and thinking that went into a particular proof step, which itself is not a lone entity but instead (usually)

follows or is supported by prior steps. In short, intent identifies the *why* of a step, which we strongly believe helps the reader to get an understanding of the *purpose* and *role* of a step.

Intent not only strengthens the structure of a proof, but makes the proof more robust. The unfortunate reality of proofs is that when they are written on paper, or without the assistance of a strict interactive proof assistant/human, they usually tend to contain bugs and typos. A proof however should be resilient to typos and small mistakes. More importantly, the larger picture and meaning of the steps should be conveyed to a reader without breaking down in the presence of these mistakes (robustness).

Moreover, we believe that expressing the flow of reasoning that an author makes when deriving a correct proof may involve pitfalls, some of which may give insight/context into why a certain strategy can't work. Sometimes, the intuition for a good proof may be absent from an actual formal proof. We believe that this intuition is crucial in understanding a proof and may be partly represented by intent (Section 6.5)). We believe that intent can encode such meta-proof information and may also fall under the category of *informal proofs* in mathematical practice philosophy [18].

Typically the way proof authors express intent in proofs is by way of natural language (e.g English). Explaining thoughts in natural language is a crucial process of making a proof more human friendly. However, here are some additional ways to encode intent in cryptographic proofs, specific to SSP methodology.

- Using good naming conventions for package/oracle/variable names.
- Visually encoding intent into SSP graphs and layouts.

6.5.1 Naming

It is a well known fact that good naming conventions can help avoid many mistakes in proof writing, whereas bad naming can lead to more mistakes and obfuscated proofs. Moreover, using meaningful names in proofs not only enables a reader to parse a proof faster, but allows the reader's understand the *intent* of author. A less obvious but useful consequence of good naming practices, with regard to SSPs, is to avoid having to parse and understand elaborate code equivalence proofs. For example, in Chapter 2 we had to show that the `Zeroer` package composed by another `Zeroer` package was code equivalent to a single `Zeroer` package. Since the name intent of this package is explained and appropriate with regard to its function, a reader may simply understand that this statement is valid without having to verify the equivalence via inlining.

6.5.2 SSP graph layouts

Intent may also be encoded effectively in SSP graph layouts. In Figure 6.1, we illustrate two such examples of encoding intent visually.

Example: Multi-instance game intent

The first example (Figure 6.1) expresses a multi-instance game intent. By laying out a call graph this way, the structure of multiple instances of a game (left of Figure 6.1) becomes apparent, since the reader is able to see multiple copies of the graph. Moreover it is clear that state is being shared between layers, by the packages that have no outgoing edges.

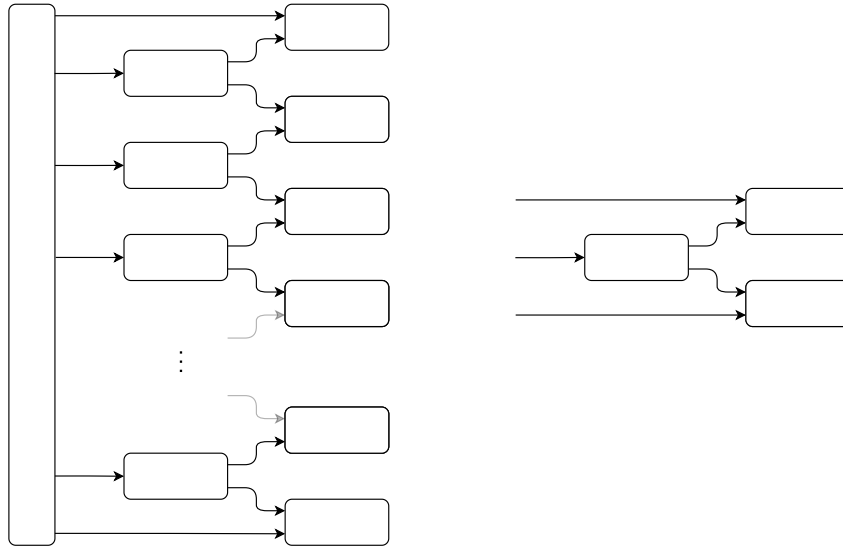


Figure 6.1: Multi-instance game intent (left) with single game (right)

Example: Rewrite intent

Additionally we could use *rewrite* intents effectively in SSP call graphs (Figure 6.2), whereby *rewriting* is the process of replacing a cut in the graph by another construction (which has matching interfaces). By laying out the next step based on the layout of previous steps, the reader is able to follow the train of thoughts that the proof author had in mind while performing these transformations.

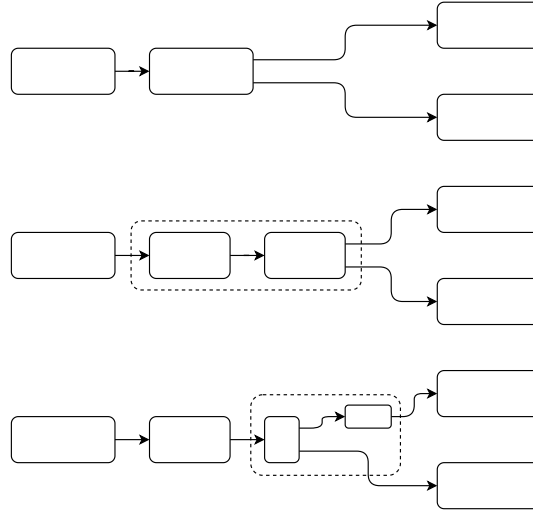


Figure 6.2: Rewrite intent

For example, we see in Figure 6.2 that step two rewrites the midsection of the call graph by introducing a new package which does not alter the behaviour of the previous system. The last step takes this further by introducing another package, but it seems to indicate that this new construction is the previous package modularized or decomposed into two other packages.

6.6 Related work

Mathematicians have always used terms to describe proofs, among them *beauty* and *elegance* to describe mathematical proofs. There has been a significant amount of effort to understand these terms precisely [18], since they play an important role in proof perception and understanding. In a recent survey by Hamami and Morris, we find that some of the concepts in mathematical practice philosophy are quite similar to concepts that have been researched by the Crypto community. One such effort is the modularization of mathematical proofs [18]. Moreover, there have been connections between proof visualization and proof cognition/understanding in the notion of informal proofs (which are a style of communicating proofs that mathematicians use to share intuition without having to be formal). We believe that such concepts such as beauty and elegance of proofs should also be given equal importance in the field of cryptography.

Chapter 7

Proof viewer for SSPs

State separating proofs and constructions are often very detailed and require space for their descriptions (both graphs and pseudocode). Moreover, SSPs are meant to be visualized. For this reason, the medium in which they appear must not be too restrictive, in which case navigating between definitions and graphs can be tedious. The main contribution of this work is to help build a suitable medium for SSPs that avoids layout constraints of a traditional, journal style paper. We provide a proof of concept (PoC) of such a proof viewer, with the proof of Yao’s garbling scheme as its primary case study.

Along the way, we develop design ideas and concepts for a SSP proof editor for writing proofs. Since this is a large undertaking, we focus only on the viewing aspect of SSPs and defer SSP proof writing to future work.

Below we identify and discuss some of the main design features of the proof viewer (Section 7.1). In Section 7.2, we discuss implementation details of the our tool¹, and motivation for some of the design choices. To test the effectiveness of these ideas in proof communication and understanding, we conducted a small User study comparing a traditional format (latex, PDF) vs the SSP proof viewer. We present the results of this study and offer an interpretation of these results (Section 7.3). Finally, we provide suggestions for how the proof viewer may be extended to achieve the large goal of proof writing and proof verification (Section 7.4).

7.1 Proof viewing and flow

viewing philosophy linear scrolling, hiding steps, pop-ups, keeping definitions besides graphs for lookup

¹We sometimes refer to the proof viewer as *the tool* throughout this chapter.

Home
Definitions
Proofs

Theorem

Main theorem.

Let \mathcal{A} be a PPT adversary, let d be a polynomial upper bound on the depth of the circuit which \mathcal{A} chooses, let n denote the width of the circuit and let sk denote the symmetric encryption scheme used within GB_{yao} . Then, there exists a PPT reduction \mathcal{R} such that

$$Adv(\mathcal{A}; SEC^c(GB_{yao}), SEC^c(SIM_{yao})) \leq d \cdot n \cdot Adv(\mathcal{A} \rightarrow \mathcal{R}; IND-CPA^s(sk), IND-CPA^s(sk))$$

$SEC^c(GB_{yao})$

$SEC^c(SIM_{yao})$

In particular, \mathcal{R} is defined as sampling a uniformly random $i \leftarrow \{1, \dots, d\}$ and running $\mathcal{R}^i := \mathcal{R}_{1, \dots, i}^i \rightarrow \mathcal{R}_{i, yao}^i \rightarrow \mathcal{R}_n$, where reduction, $\mathcal{R}_{1, \dots, i}^i$ is defined in Lemma 2, reduction $\mathcal{R}_{i, yao}^i$ is in Lemma 1 and reduction \mathcal{R}_n is defined in Lemma 3.

Lemma 1

Lemma 1 (circuit security).

Let d be a polynomial upper bound on the depth of the circuit which \mathcal{A} chooses. Then for each $1 \leq i \leq d$ there exists PPT reductions, $\mathcal{R}_{i, yao}^i$ such that

$$Adv(\mathcal{A}; SEC^c(GB_{yao}), SEC^c(SIM_{yao})) \leq \sum_{i=1}^d Adv(\mathcal{A} \rightarrow \mathcal{R}_{i, yao}^i; LSEC^c(GB_{yao}), LSEC^c(SIM_{yao}))$$

KEYS

<p>SETBIT(j, z)</p> <pre> assert $z_j \neq \perp$ $z_j \leftarrow z$ return () </pre>	<p>GETBIT(j)</p> <pre> assert $z_j \neq \perp$ return z_j </pre>	<p>GETA^{out}(j)</p> <pre> assert $z_j \neq \perp$ aflag $\leftarrow 1$ if $Z_j = \perp$ then $Z_j(0) \leftarrow \{0, 1\}^k$ $Z_j(1) \leftarrow \{0, 1\}^k$ return $Z_j(z_j)$ </pre>
<p>GETAⁱⁿ(j)</p> <pre> assert $z_j \neq \perp$ assert aflag = 1 assert $Z_j \neq \perp$ return $Z_j(z_j)$ </pre>	<p>GETKEYSⁱⁿ(j)</p> <pre> assert $z_j \neq \perp$ return z_j </pre>	<p>GETKEYS^{out}(j, z)</p> <pre> assert $z_j = \perp$ $z_j \leftarrow z$ return () </pre>

MODGB

$GBL(\ell, r, op, j)$

```

assert  $C \neq \perp$ 
assert  $\ell, r, op \neq \perp$ 
assert  $|\ell|, |r|, |op| = n$ 
for  $j = 1..n$  do
   $(\ell, r, op) \leftarrow (\ell(j), r(j), op(j))$ 
   $C_j \leftarrow DENC(\ell, r, op)$ 
 $\hat{C} \leftarrow \hat{C}_{j..n}$ 
return  $\hat{C}$ 
                    
```

MODDENC

$DENC(\ell, r, op, j)$

```

 $\hat{b}_j \leftarrow \perp$ 
 $Z_j^{out} \leftarrow GETKEYS^{out}(j)$ 
for  $(b_1, b_2) \in \{0, 1\}^2$  do
   $b_j \leftarrow op(b_1, b_2)$ 
                    
```

Figure 7.1: Screenshot of SSP proof viewer.

Navigation and linking

Navigating a proof is often inconvenient in the traditional (journal style paper) format, and requires a lot of scrolling. We would like to eliminate unnecessary scrolling to find definitions and to get parts of the proof that one is interested in. To establish effective proof navigation in the proof viewer, we employ the concept of “linking” in as many ways as possible. Here are some desired types of linking:-

1. Links from the proof tree to a claim/lemma
2. Links from a proof step to another proof step
3. Links of pseudocode oracles

Jumping ahead, since we implement the proof viewer as a dynamic website (in JavaScript and HTML), linking is a natural property that we get for free, achieving desired linking (2) and (3). However, the main device/feature for navigating to statements and steps of a proof, we implement a dynamic proof tree. Every node in this dynamic proof tree is either a proof step (reduction/code equivalence) or a statement (e.g. theorem/lemma/claim). The proof tree for Yao can be found in the bottom right corner of Figure 7.1. Clicking on a node of the proof tree brings the reader to the desired proof step on the left pane.

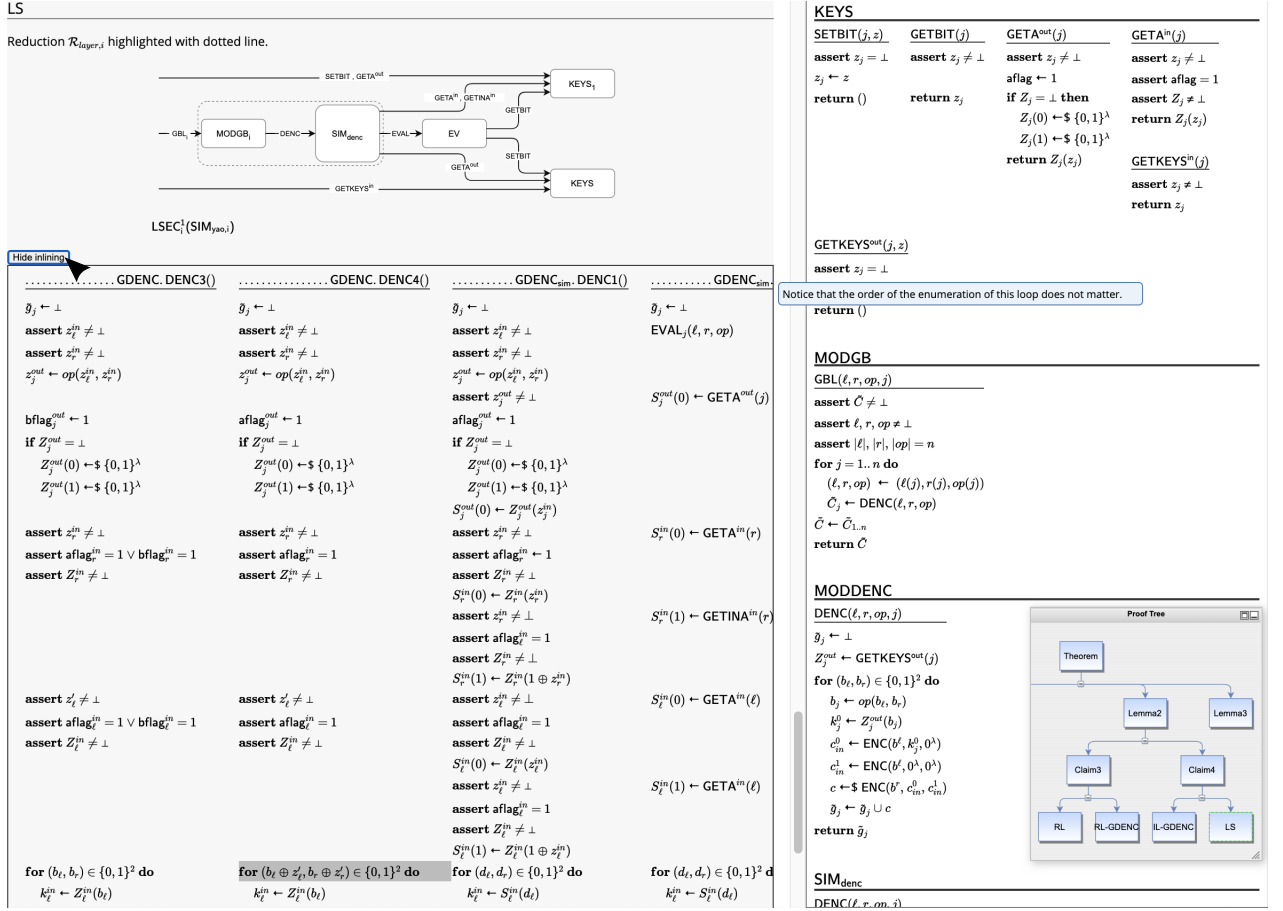


Figure 7.2: Inlining steps in the proof viewer.

Information hiding

We integrate several features for information hiding in the proof viewer. Here are some of them (depicted also in Figure 7.1).

1. *Hiding proof statements/step(s)*. To achieve this, we allow the reader to close certain parts of the proof by collapsing the parts of the proof tree, namely every node provides an option to remove its children from the main proof pane.
2. *Hiding code equivalence/inlining steps*. The proof viewer hides code equivalence/inlining steps by default. The reader is given the option to click and reveal the step (cursor in Figure 7.2) if desired, and is able to view the inlining steps without the other pseudocode pane changing, allowing comparisons to not get “lost” in viewing details.

3. *Hiding text and explanations.* Another feature for better viewing inlining step explanations is to have annotations or popups that appear only when a certain line in the pseudocode is being hovered over. In Figure 7.2 we see an example of a annotation to the right of main viewing pane (albeit it is rather small in the screenshot). An additional feature of the proof viewer is to hide all text entries, leaving only SSP call graphs. This would be useful for a reader who is interested in verifying most of the syntactic statements since this is mostly encoded visually in SSP.

7.2 Implementation details

7.2.1 Overview of design goals and choices

Below we present some (broader) design goals with regard to the proof viewer, and how we achieved them.

- *Encourage people to contribute.* To ensure a low learning curve, the proof viewer has been designed with simplicity in mind. We restricted usage of fancy frameworks, except where necessary, with almost all of the code written entirely in JavaScript.
- *Make the tool accessible and un-intrusive.* At present the proof viewer does not require any installation, as it runs in a browser, making it accessible to most people (with computers). It is an important design consideration to not force the user to use something that it doesn't want to. One way we have tried to solve this is by minimizing the number of dependencies the tool uses, as well as relying only on open source frameworks, such as mxGraph. Moreover, the proof viewer is designed to be run offline, the idea being that the user may switch off the Internet if it wanted to and still be able to work uninterrupted. Unfortunately, at present, the proof viewer requires internet access for rendering L^AT_EX, but we hope to solve this in the future.

Handling SSP graphs

To enable faster creation and manipulation of SSP graphs by a proof author, we created a basic editor that takes as input a graph represented in JSON format and produces an SSP call graph in SVG format that can be edited further in any internet browser.

Proof format

We also use JSON as the language format for proofs. At the top level, a proof JSON file contains 4 properties, a name, monolithic package definitions, modular package definitions (unlike monolithic packages, these contain graph information since they represent packages that are constructions) and a proof tree (Figure 7.3).

```

1 {
  "name": "...",
  "monolithic_pkgs": {...},
  "modular_pkgs": {...}
  "prooftree": {...},
}

```

Figure 7.3: properties of proof file

```

"BITS":
{
  "oracles":
  {
    "CHECK" :
    {
      "code": "@assert z_j \\\neq @bot;",
      "params": ["j"]
    },
    "GETBIT" :
14  {
      "code": "@assert z_j \\\neq @bot; ;@return z_j",
      "params": ["j"]
    },
    "SETBIT" :
    {
      "code": "@assert z_j = @bot;z_j @gets z;@return ();",
      "params": ["j", "z"]
    }
  }
},
"BITS_1":
{
  "instance": "BITS"
},
34 "BITS_2":
{
  "instance": "BITS"
},

```

Figure 7.4: Example of monolithic package and instances (BITS package) definition.

In Figure 7.4, we see how a (monolithic) package is defined. The description allows an author to add package instances as well, because our

```

3 {
  "SEC^0(GB_{yao})":
  {
    "oracles": [[["KEYS_1", "SETBIT"], ["KEYS_1", "GETA^{out}"],
                 ["GB_{yao}", "GBL_{1..d}"], ["KEYS", "GETKEYS^{in}"]],
    "graph":
    {
      "GB_{yao}": [[["KEYS_1", "GETKEYS^{in}"], ["KEYS", "GETKEYS^{out}"]],
      "KEYS_1": [],
      "KEYS": []
    },
    "layout": {"nodes": {...}, "edges": {...}, "edge_points": {...}
  },
}

```

Figure 7.5: Example of modular package definition with graph data structure.

graph data structure requires unique package names for every node in the call graph. At present the package indices are left in the visualization, but this can be removed also. Also notice that the for monolithic packages we may want oracle definitions. In Figure 7.4 we see the oracle definitions of CHECK, GETBIT and SETBIT.

On the other hand, modular packages are essentially describe constructions, hence they require graph information, which we represent with an adjacency list in JSON (Figure 7.5). We have omitted layout information in this figure, but in practice this information is anything pertaining to the *layout* of the graph, such as package coordinates and edge/arrow placements.

In Figure 7.6 we see how a proof tree is defined in JSON. Each step has some content that is either text or graphs. The layout of graphs is implicitly specified in the way that the list defined, e.g. $[[A, B]]$ would put graphs A and B side by side, whereas $[[A], [B]]$ would make graph A go above graph B .

7.3 User study

To test the effectiveness of these ideas and the proof viewer's implementation of them, we conducted a user study. The study was conducted on a small group of 6 students, who had taken a graduate level course on Cryptography prior to the study. The participants were split into into 2 groups consisting of 3 people each. Participants were not allowed to have any discussion among themselves. Group 1 was selected as the constant group, and was given a traditional style paper presenting the proof of Yao's garbling scheme. On the other hand, Group 2 was the test group, with which we provided the same proof but in the proof viewer. To familiarize the participants with the garbled circuits protocol, the participants were provided a short document

```

5  "Theorem" :
    {
      "parent": null,
      "contents": [
        {
          "text": "Main theorem."
        },
        {
          "text": "Let  $\mathcal{A}$  be a PPT adversary, let  $d$  be a polynomial upper bound
on the depth of the circuit which  $\mathcal{A}$  chooses, let  $n$  denote the
width of the circuit and let  $se$  denote the symmetric encryption scheme used
within  $\mathcal{GB}_{\text{yao}}$ . Then, there exists a PPT reduction  $\mathcal{R}$ 
such that  $\Pr[\text{Adv}(\mathcal{A}; \text{SEC}^0(\mathcal{GB}_{\text{yao}})), \text{SEC}^1(\text{SIM}_{\text{yao}})] \leq d \cdot n \cdot \Pr[\text{Adv}(\mathcal{A}) \rightarrow \mathcal{R}; \text{IND}_{\text{CPA}^0}(se), \text{IND}_{\text{CPA}^1}(se)]$ "
        },
        {
          "graphs": [{"SEC^0(GB_{yao})"}, {"SEC^1(SIM_{yao})"}, {"LSEC^1_{1..d}(GB^1_{yao})"}]
        },
        {
          "text": "...
        }
      ]
    },
25  "Lemma1" :
    {
      "parent": "Theorem",
      "contents": [
        {
          "text": "Lemma 1 (circuit security)."
        },
        {
          "text": "...
        }
      ]
    },
45  "Lemma2" :
    {
      "parent": "Theorem",
      "contents": [
        {
          "text": "Lemma 2 (layer security)."
        },
        {
          "text": "...
        }
      ]
    },
    ...

```

Figure 7.6: Proof tree JSON description.

(a week prior to the study) that gave an overview of garbled circuits.

7.3.1 Setup and questions

The questions were divided roughly into 3 parts, namely, structural, technical and subjective questions (Appendix B). In the structural questions, we tried to get an idea of the participants' understanding of the structure of the proof and its statements, i.e. the lemmas and claims of the proof. In the technical questions, we gauged their understanding of low level specifics, by quizzing

them on details within the pseudocode of oracles.

7.3.2 Interpretation of results

Structural questions

We found that the proof viewer enhanced participants' understanding of proof structure. This may be an unfair comparison, since the proof viewer has an explicit proof tree whereas the paper had encoded it in a slightly different way. Nevertheless, this shows that explicitly having a proof tree, which also acts as a "map" for navigating the proof reinforces the understanding of a proof's structure.

Technical questions

Participants exposed to the proof via the proof viewer seemed to also perform better on technical questions pertaining to pseudocode. One reason for this could be that the oracle pane/window is always beside the proof steps, and is easy to access by clicking on packages within a call graph. Moreover, we provided links in oracle pseudocode to the other oracles it makes calls to, which one participant regarded as useful.

Feedback and improvements

The participants that interacted with the proof viewer mainly felt that they were unable to understand certain steps. This was mainly due to the fact that the proof was incomplete. However, with regard to visualizations - both groups felt that it helped them to understand the high level structure and idea of the steps (keep in mind that they were only given 1 hour and 45 minutes for this entire experiment), and we feel this is a positive result indicating the validity of some of our hypotheses on proof viewing.

7.4 Future work

In general, we may conclude that the proof viewer is indeed a useful tool for better proof communication (a positive result!). Below we list some natural extensions to the SSP proof viewer.

Improvements to the proof viewer

- **Better visualization.** For example, hybrids proof steps can be colored with a gradient, reduction steps can be highlighted.
- **Improved automatic graph layouts.** We would like to use better/-more advanced algorithms to generate better graph layouts automatically. The current version of the proofviewer simply converts each layer of the SSP call graph DAG and displays it on a separate column, with some pre-defined offset.
- **Improved interface.** This is an open problem. There are probably other (better) ways to lay out the interface and viewports.

A proof editor for SSPs

The initial goal of this project was to build a full-fledged proof editor. We would like to re-iterate that this would be a very useful tool, since it would allow proof authors to also write better proofs (less mistakes, and more efficiently).

- **Graph rewriting and interface matching.** However, it would be good to somehow integrate the *pseudocode* of oracles along with graphs to see if they are valid/correct.
- **A richer graph editor.** At present the proof viewer is a bit dull (highlighting reductions with gray and code equivalences with dashed lines). It would be useful to include color information or shapes.
- **IPython-Notebook style editing of proofs.** Seldom do proof authors immediately come up with a correct proof and write it in a single iteration. Notebook style editing will allow authors to experiment and try different strategies, while being able saving steps that have been tried, and to have view partially worked steps.
- **Collaborative proof writing.** This is an advanced feature, but we hope someday cryptographers will be able to collaborate and write proofs in realtime via the Internet.

Automated code equivalence and parameterization in the language

Code equivalence steps can be quite tedious, especially when they involve multiple package composition. Also, using pseudocode gives the author more

freedom but comes with the downside that it cannot be machine checked, and hence may be error-prone. We hope these steps can be avoided in the first place, but this will ultimately rely on well designed constructions for the proof are, and effective application of laying out, naming, intent and so on.

The other solution to avoid code equivalence proofs is to automate them. In order to automate code equivalence steps the author must use instead parameterize the language of the oracles with some formal programming language instead of pseudocode.

7.5 Conclusion

Much of the results of this project was due to the author's (and surrounding community's) desire for better tools in cryptographic proof communication and understanding. In the early stages of this project, the main objective was to understand garbled circuits, and its proof of security, through machine verification methods. However, as the research evolved, several other questions emerged i.e. methods of better cryptographic proof communication. Our hope is that this project may inspire more people to research and study better proof communication techniques.

Bibliography

- [1] ABRIL, V. A., MAENE, P., MERTENS, N., AND SMART, N. Bristol fashion mpc circuits, 2019.
- [2] APPLEBAUM, B. Garbled circuits as randomized encodings of functions: a primer. In *Tutorials on the Foundations of Cryptography*. Springer, 2017, pp. 1–44.
- [3] BALL, M., CARMER, B., MALKIN, T., ROSULEK, M., AND SCHIMANSKI, N. Garbled neural networks are practical. *IACR Cryptology ePrint Archive 2019* (2019), 338.
- [4] BARTHE, G., DUPRESSOIR, F., GRÉGOIRE, B., KUNZ, C., SCHMIDT, B., AND STRUB, P.-Y. Easycrypt: A tutorial. In *Foundations of security analysis and design vii*. Springer, 2013, pp. 146–166.
- [5] BEAVER, D., MICALI, S., AND ROGAWAY, P. The round complexity of secure protocols. In *Proceedings of the twenty-second annual ACM symposium on Theory of computing* (1990), pp. 503–513.
- [6] BELLARE, M., HOANG, V. T., AND ROGAWAY, P. Foundations of garbled circuits. In *Proceedings of the 2012 ACM conference on Computer and communications security* (2012), pp. 784–796.
- [7] BELLARE, M., AND ROGAWAY, P. The security of triple encryption and a framework for code-based game-playing proofs. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques* (2006), Springer, pp. 409–426.
- [8] BEN-OR, M., GOLDWASSER, S., AND WIGDERSON, A. Completeness theorems for non-cryptographic fault-tolerant distributed computation. In *Providing Sound Foundations for Cryptography: On the Work of Shafi Goldwasser and Silvio Micali*. 2019, pp. 351–371.

- [9] BLANCHET, B. Automatic verification of security protocols in the symbolic model: The verifier proverif. In *Foundations of Security Analysis and Design VII*. Springer, 2013, pp. 54–87.
- [10] BLEICHENBACHER, D. Chosen ciphertext attacks against protocols based on the rsa encryption standard pkcs# 1. In *Annual International Cryptology Conference (1998)*, Springer, pp. 1–12.
- [11] BRZUSKA, C., DELIGNAT-LAVAUD, A., KOHBROK, K., AND KOHLWEISS, M. State-separating proofs: A reduction methodology for real-world protocols. *IACR Cryptology ePrint Archive 2018* (2018), 306.
- [12] BRZUSKA, C., AND OECHSNER, S. State-separating constructions: The case of yao’s garbling scheme. Manuscript.
- [13] CANETTI, R. Universally composable security: A new paradigm for cryptographic protocols. In *Proceedings 42nd IEEE Symposium on Foundations of Computer Science (2001)*, IEEE, pp. 136–145.
- [14] CHOU, T., AND ORLANDI, C. The simplest protocol for oblivious transfer. In *International Conference on Cryptology and Information Security in Latin America (2015)*, Springer, pp. 40–58.
- [15] DEVADAS, S., AND LEHMAN, E. Lecture notes in mathematics for computer science, February 2005.
- [16] GHAZI, B., KOMARGODSKI, I., KOTHARI, P. K., AND SUDAN, M. Communication with contextual uncertainty. *computational complexity* 27, 3 (2018), 463–509.
- [17] HALEVI, S. A plausible approach to computer-aided cryptographic proofs. *IACR Cryptol. ePrint Arch. 2005* (2005), 181.
- [18] HAMAMI, Y., AND MORRIS, R. Philosophy of mathematical practice: A primer for mathematics educators. *ZDM-MATHEMATICS EDUCATION* (2020).
- [19] IMPAGLIAZZO, R. A personal view of average-case complexity. In *Proceedings of Structure in Complexity Theory. Tenth Annual IEEE Conference (1995)*, IEEE, pp. 134–147.
- [20] JAFARGHOLI, Z., AND OECHSNER, S. Adaptive security of practical garbling schemes. Cryptology ePrint Archive, Report 2019/1210, 2019.

- [21] KOLESNIKOV, V., MOHASSEL, P., AND ROSULEK, M. Flexor: Flexible garbling for xor gates that beats free-xor. In *Annual Cryptology Conference* (2014), Springer, pp. 440–457.
- [22] KOLESNIKOV, V., AND SCHNEIDER, T. Improved garbled circuit: Free xor gates and applications. In *International Colloquium on Automata, Languages, and Programming* (2008), Springer, pp. 486–498.
- [23] LAMPORT, L. Constructing digital signatures from a one-way function. Tech. rep., Technical Report CSL-98, SRI International, 1979.
- [24] LINDELL, Y., AND PINKAS, B. A proof of security of yao’s protocol for two-party computation. *Journal of cryptology* 22, 2 (2009), 161–188.
- [25] LIU, J., JUUTI, M., LU, Y., AND ASOKAN, N. Oblivious neural network predictions via minion transformations. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (2017), pp. 619–631.
- [26] LOWE, G. Breaking and fixing the needham-schroeder public-key protocol using FDR. *Softw. Concepts Tools* 17, 3 (1996), 93–102.
- [27] MAKRI, E., ROTARU, D., SMART, N. P., AND VERCAUTEREN, F. Epic: Efficient private image classification (or: Learning from the masters). In *Cryptographers’ Track at the RSA Conference* (2019), Springer, pp. 473–492.
- [28] MAURER, U. Constructive cryptography—a new paradigm for security definitions and proofs. In *Joint Workshop on Theory of Security and Applications* (2011), Springer, pp. 33–56.
- [29] MEIER, S., SCHMIDT, B., CREMERS, C., AND BASIN, D. The tamarin prover for the symbolic analysis of security protocols. In *International Conference on Computer Aided Verification* (2013), Springer, pp. 696–701.
- [30] NAOR, M., PINKAS, B., AND SUMNER, R. Privacy preserving auctions and mechanism design. In *Proceedings of the 1st ACM conference on Electronic commerce* (1999), pp. 129–139.
- [31] PINKAS, B., SCHNEIDER, T., SMART, N. P., AND WILLIAMS, S. C. Secure two-party computation is practical. In *International Conference on the Theory and Application of Cryptology and Information Security* (2009), Springer, pp. 250–267.

- [32] ROSULEK, M. The joy of cryptography. *Oregon State University EOR* (2018), 1.
- [33] SHANNON, C. E. Communication theory of secrecy systems. *Bell system technical journal* 28, 4 (1949), 656–715.
- [34] SHOUP, V. Sequences of games: a tool for taming complexity in security proofs. *IACR Cryptol. ePrint Arch. 2004* (2004), 332.
- [35] YAO, A. C.-C. How to generate and exchange secrets. In *27th Annual Symposium on Foundations of Computer Science (sfcs 1986)* (1986), IEEE, pp. 162–167.
- [36] ZAHUR, S., ROSULEK, M., AND EVANS, D. Two halves make a whole. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques* (2015), Springer, pp. 220–250.

Appendix A

Attack on deterministic encryption scheme

To see this why deterministic encryption is not IND-CPA secure, let us instead of using se , use a deterministic encryption scheme $detse$. Consider the following adversary (Figure A.1):

```
 $\mathcal{A}$ 

---

1 :  $c_0 \leftarrow \text{ENC}(0^\lambda)$   
2 :  $c_1 \leftarrow \text{ENC}(1^\lambda)$   
3 : if  $c_0 = c_1$  then  
4 :   return 1  
5 : return 0
```

Figure A.1: Adversary for deterministic encryption scheme $detse$

Essentially what the adversary does is encrypt two unique messages by calling the game's oracle ENC with λ many zeroes 0^λ , to obtain c_0 , and then with a different message 1^λ to obtain c_1 . If the ciphertexts are equivalent, the adversary knows that it is interacting with the ideal game, since the ideal game always yields an encryption of all zeroes. If it was interacting with the real game the ciphertexts would have been different.

Appendix B

User study

B.1 Questions

B.1.1 Structural questions

1. Which statement establishes that the *layer* security of Yao's garbling scheme can be reduced to double encryption security?
2. Which statement implies that the layer security of Yao's garbling scheme implies the security of the garbling of the entire circuit?
3. In the proof of Lemma 1, what do the game hops do?

B.1.2 Technical questions

1. How is state shared between layers? What packages are used to ensure this shared state?
2. (a) Is the adversary allowed to call GBL before setting an input?
(b) Can GBL be called twice?
(c) How is the order of oracle calls enforced in the proof?
(d) How and why is `aflag` used?
3. (a) In the IND-CPA game, what happens if we add a `GETINAout` query to the KEYS package so that the adversary can call it? You may provide a high level intuition/explanation or pseudocode to support your argument.

(b) If the security definition remains meaningful, explain why. If not, give an attack which has distinguishing advantage $\geq \frac{3}{4}$.

B.1.3 Subjective questions

1. Which part of the proof did you least understand?
2. Which part of the proof did you most understand?
3. Out of a scale of 1 to 10, how well would you say you understood the pseudocode used in the packages?
4. Out of a scale of 1 to 10, how well would you say you understood how the proof steps come together (typically these are found in Lemmas and Claims)?
5. Did you find the visualizations of the call graph helpful in understanding the proof? If so please state which components were helpful. If not, please share suggestions to improve them.